

版权相关注意事项：

- 1、书籍版权归著者和出版社所有
- 2、本PDF来自于各个广泛的信息平台，经过整理而成
- 3、本PDF仅限用于非商业用途或者个人交流研究学习使用
- 4、本PDF获得者不得在互联网上以任何目的进行传播
- 5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
- 6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
- 7、请于下载PDF后24小时内研究使用并删掉本PDF

从微服务的架构设计模式和技术造型入手，以Spring Boot 2、Spring Cloud和Docker为构建框架，实现横向可扩展的高可用架构



微服务架构实战

张锋 / 著

洞悉微服务的构建流程，从实战的角度介绍微服务使用的关键框架
依据敏捷开发的原则，快速迭代，以完整的示例实现整个CI/CD的流程，快速响应需求



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

关于作者

张 锋



北京航空航天大学软件工程硕士，资深架构师，有十多年管理和架构经验，在业界颇具威望和影响力。

曾就职于神州数据、亚信科技、中文在线及多家互联网公司，担任架构师及技术总监等职位，现就职于中青旅，任架构组组长，曾成功管理和指导三农综合服务信息平台、西北企业云服务平台、省级电信平台及多个互联网平台的架构升级改造。

拥有工信部认证高级信息系统项目管理师资格。

博客园推荐博客，阿里云社区认证专家，腾讯云社区认证专家。





微服务架构实战

张锋 / 著

电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING



内 容 简 介

目前微服务的现状是概念多于实践,管理和架构人员往往不知道如何落地微服务,本书从软件工程的角度切入,融入了作者多年的管理及架构经验,内容完全基于实际经验所得,直击痛处。通过阅读本书,开发人员能够实现微服务的快速落地。

全书共 12 章,分为概念篇、开发篇、运维篇和实战篇。概念篇详细阐述微服务的由来和设计要点。开发篇介绍 Spring Boot、Docker 和 Spring Cloud 应用于微服务的案例,并且附有源代码。运维篇从测试、快速开发、质量管理、自动化运维和监控的角度介绍微服务涉及的知识点。实战篇将企业级开发中涉及的内容尽可能详细地列出。

本书不但适合初学者,而且对于团队的管理者及技术选型的架构师也有着非常大的参考意义。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有,侵权必究。

图书在版编目(CIP)数据

微服务架构实战 / 张锋著. —北京: 电子工业出版社, 2018.6
ISBN 978-7-121-34342-1

I. ①微… II. ①张… III. ①互联网络—网络服务器 IV. ①TP368.5

中国版本图书馆 CIP 数据核字(2018)第 120574 号

责任编辑: 陈晓猛

印 刷: 三河市良远印务有限公司

装 订: 三河市良远印务有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱

邮编: 100036

开 本: 787×980 1/16 印张: 25.5

字数: 492 千字

版 次: 2018 年 6 月第 1 版

印 次: 2018 年 6 月第 1 次印刷

定 价: 89.00 元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888, 88258888。

质量投诉请发邮件至 zltz@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式: 010-51260888-819, faq@phei.com.cn。



专家评价

(排名不分先后)

架构的演进是循序渐进并非一蹴而就的，随着时代的更替，我们对于不同时期、阶段的技术发展，也有着不一样的理解和选择。在互联网+和大数据的时代下，如今的软件架构已经从单体、垂直、SOA，发展到微服务架构。本书的作者是业内资深架构师，有着多年的架构从业经验，常年活跃在 Spring Cloud 社区，致力于企业级微服务的解决方案和落地实现。全书以简洁、循序渐进的风格，由浅入深，从易到难，清晰地向读者介绍了微服务的技术思想和整体生态，不仅适合初学者入门，也适合有经验的人员进阶，值得一看。

美团无人配送部

高级技术专家

刘审川 博士

本书以实践为主，内容涵盖了微服务的整个生态，详细解释了实践微服务必须要面对的架构模式。

本书系统地介绍了微服务的设计、开发、运维等方面，结合了 Spring Boot 和 Docker 等热点技术，对微服务的整个生命周期做了全面介绍，适合对微服务实践感兴趣，以及想成为微服务架构师的人员阅读。

当当网

技术总监 穆幽方

为适应市场的变化，企业自身的技术需要不断迭代。如何减少频繁迭代带来的开销成为架构师、开发和运维等人不断思考的问题，“微服务”似乎成为了一个非常好的选择。本书全面地介绍了微服务相关知识。

作者从基本概念谈起，循序渐进地介绍了各种实用技术，无论你处在技术链中的哪个环节，都能找到对应的知识点并快速学习，同时能了解其他环节的工作内容，值得技术爱好者收藏和学习。

中文在线

运维经理 姜楠



随着微服务的兴起，我们可以看到微服务正逐步应用在企业中，与之带来的改变不言而喻，微服务的独立部署、可扩展性架构等特点可以帮助企业提高生产效率、节省成本。本书围绕一个中心，从微服务的概念、架构、管理、应用等方面层层深入，并通过翔实的案例结合实践进行展开，能够助力企业快速实施与部署微服务，值得一读。

柏睿数据科技（北京）有限公司
产品总监 韩辉辉

在软件工程开发的技术序列里，常常是极强必简。《微服务架构实战》就是这种用简洁的论述解决复杂问题的好书。作者张锋从实践中来到实践中去，将强大的微服务开发武器，用简约易懂的方式展现在读者面前，值得广大求道而解惑的技术人员深入领会。

丽华数据分析引擎工作室
技术总监 吉更

微服务目前已经成为主流互联网技术架构，在处理复杂业务解耦的问题上屡试不爽。本书作者张锋长期活跃在 Spring Cloud 社区，致力于帮助企业完成微服务改造，并提供咨询服务。本书作为全面的企业微服务架构实践指南，不仅可以帮助初识微服务的开发者完成微服务架构的搭建，也可以帮助正在实践微服务架构的团队解决实际问题。作者长期从事技术类博主的撰写，擅于深入浅出地讲解枯燥晦涩的技术细节，娴熟的文笔在很大程度上可以帮助读者轻松掌握书中的知识点。

北京环宇万维科技有限公司
技术经理 苏忠亮

当下，越来越多的企业选择与互联网深度融合，基于互联网搭建信息化平台。在平台搭建的过程中，必然要考虑到后期的日常运营带来的爆炸式数据增长所催生出的如高并发、秒级响应速度、分布式部署等一系列问题。微服务的应用正是应对这类严峻考验的极佳解决方案。本书是微服务领域比较完善的实践探索经验的结晶，从微服务的思想基础、设计原则，衍生到 Spring Boot、Docker、Spring Cloud 及其他框架的介绍，不仅涉及微服务的自动化测试与质量管理、自动化部署、日志收集与监控，还提供完整的实战示例，并进行了核心功能推荐的全方位介绍。全书清晰、透彻地剖析了微服务的整个生态，书中还配有許多系统插图，有助于读者快速提升对微服务的认知，构建自己的架构体系。此书具有极高的可读性和应用价值，可谓开卷有益，是践行微服务的上佳选择。

北京亿心宜行汽车技术开发服务有限公司技术部
高软件开发工程师 邓威



前言

从分布式服务到 SOA，再到微服务，服务化的脚步一直在不断地前进。正所谓“分久必合，合久必分”，在企业高速发展的今天，单体架构已经很难适应业务的快速变化，微服务的出现，为应对快速变化的业务需求、冗长的开发周期提供了一种新的解决方案。它以模块化的思维应对快速变化的业务需求，使用比如自动化部署、自动化业务监控预警、调用链监控、容器化，以及快速开发等思想加快软件的开发周期，实现更快速、更高质量的交付，整体提高客户的满意度。

内容介绍

本书系统地介绍了微服务涉及的各种知识点，横跨软件开发的整个生命周期，采用目前前沿的技术进行知识点的展开。微服务是一个概念，就像 SOA 一样，可能在不同的环境中会产生不同的设计方案。但是总的来说，微服务是为了解决高并发、大数据量的问题而产生的分布式的综合系统解决方案。

本书的内容安排非常有层次感，对于软件开发和从业人员从整体上了解和掌握微服务所需要的知识点进行了全面的梳理。

本书可以分为概念篇、开发篇、运维篇和实战篇。

概念篇

首先从概念的角度出发，让读者对微服务的发展有一个感观的了解，然后从设计理念上给出一些建议。

第 1 章从微服务的起源和现实业务的角度探讨微服务，使读者能够对微服务有一个感观的认识。

第 2 章是针对微服务的设计理念进行整理，包括服务如何拆分、前后端分离、CAP 理论和 CQRS 等，是一个高层次的指导原则。



开发篇

开发篇以 Java 中常用的微服务框架 Spring Boot 为基础，介绍 Spring Boot 的快速开发，以及 Docker 技术的基础，并且完成两者的无缝结合。接着对 Spring Cloud 的整体架构进行介绍。

第 3 章详细地介绍 Spring Boot 的开发，包括使用它的优缺点，以及在企业级开发中常用的工具包的整合，包括面向切面编程、Web 开发、文档管理和调度管理，最后结合 Dubbo 完成一个示例性的分布式工程。

第 4 章主要讲解 Docker 的基础操作，介绍微服务中所用到的容器相关的技术，最后给出通用的基于容器的私有云架构。

第 5 章对 Spring Cloud 实现微服务的几个重要框架进行展开描述，让读者了解注册中心、负载均衡、容错、分布式配置、网关和消息总线，能够完成开发层面的微服务架构。

第 6 章对 Spring Cloud 的非核心框架进行介绍，包括 Consul、ZooKeeper、安全框架和数据流框架。

通过对以上几章的了解，读者应该能够从开发的角度基本掌握微服务的开发。

运维篇

在微服务中，涉及的不仅仅是开发，还会涉及很多的点，包括运维、测试、监控和日志管理。

第 7 章主要对测试和质量管理进行介绍，测试部分包括单元测试、A/B 测试、冒烟和回归测试，质量管理部分主要使用静态代码分析，并且基于 SonarQube 对代码进行静态检查，以及分析代码的总体质量。

第 8 章对微服务的最佳实践 JHipster 进行系统的介绍，并且对 JHipster 部分内容做了处理，将在国内不是很流行的部分进行了处理，尽可能详细地介绍 JHipster 的应用和配置。

第 9 章主要对自动化部署进行介绍，因为微服务的目的不仅仅是简化开发，而且能够提高整个团队的运行效率。所以私服的使用和自动化运维就显得非常重要。

第 10 章主要讲解日志收集和 APM 监控，对于线上系统来说，出现问题的概率还是非常大的，如何快速定位并第一时间找到问题所在的点就显得非常重要。APM 部分对常用的监控工具进行列举，重点介绍 Pinpoint，对使用和邮件告警也进行了重点介绍。

通过以上几章的了解，读者应该能够充分理解 DevOps 的概念，并且了解微服务并不单纯是开发人员的工作，而是整个团队的协同合作。



实战篇

第 11 章通过对 PiggyMetrics 的全面讲解,让读者能够了解一个简单的微服务架构所包含的技术点和构建原则,并且实际部署微服务,完成业务的基础操作。

第 12 章对在微服务构建过程中可能涉及的技术点进行讲解,包括工作流引擎、规则引擎、调度系统、分布式配置及单点登录。

通过以上几章的学习,读者应该能够在技术选型的过程中扩展思路,了解更多的分布式业务涉及的扩展知识,并且有选择地应用到业务中。

勘误和支持

随着技术的进步,微服务的架构也会有不同的演化,就像 Spring 5 的发布一样,为前后端分离进一步奠定了基础。本书中涉及的内容大多是个人的理解和认知,难免有不足之处,所以本书中提及的知识点主要作为抛砖引玉之用,如果有错误之处,还请读者指正。

由于笔者水平有限,且编写时间仓促,书中难免会出现一些错误或者不准确的地方,恳请读者批评指正。如果您有问题或者宝贵意见,欢迎发送邮件至邮箱 cloudskyme@163.com,期待能够得到您的反馈。

致谢

首先要感谢我的家人,没有他们默默的支持,我不可能坚持完成本书的创作。

然后要感谢电子工业出版社博文视点的陈晓猛等编辑对本书内容的校对、勘正及反复核对,感谢你们的付出。

最后感谢在写作过程中给予我帮助的朋友们。

张 锋



读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **下载资源：**本书如提供示例代码及资源文件，均可在 [下载资源](#) 处下载。
- **提交勘误：**您对书中内容的修改意见可在 [提交勘误](#) 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动：**在页面下方 [读者评论](#) 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/34342>



目录

第 1 章 微服务概述	1
1.1 什么是微服务	2
1.2 为什么微服务	3
1.2.1 区别	4
1.2.2 从单体应用说起	5
1.2.3 第一步切分	5
1.2.4 服务化所带来的问题	6
1.2.5 微服务的可扩展性	8
1.2.6 微服务与 SOA 的区别	8
1.3 常见的微服务组件	9
1.4 常用的微服务框架	11
1.4.1 Spring 顶级项目	11
1.4.2 背景	12
1.4.3 社区活跃度	14
1.4.4 架构完整度	16
1.4.5 总结	17
1.5 微服务架构设计模式	17
1.6 如何实施微服务	20
1.7 小结	20
第 2 章 微服务设计原则	21
2.1 设计原则之分层架构	22
2.2 设计原则之统一通信协议	24

2.3	设计原则之单一职责	25
2.4	设计原则之服务拆分	26
2.5	设计原则之前后端分离	28
2.6	设计原则之版本控制	29
2.7	设计原则之围绕业务构建.....	30
2.8	设计原则之并发流量控制.....	30
2.9	设计原则之 CAP	31
2.10	设计原则之 EDA 事件驱动	32
2.11	设计原则之 CQRS	33
2.12	设计原则之基础设施自动化.....	35
2.13	设计原则之数据一致性.....	35
2.14	设计原则之设计模式	36
2.15	设计原则之 DevOps	39
2.16	设计原则之无状态服务.....	40
2.17	小结	41
第 3 章	微服务之 Spring Boot	42
3.1	一切从简单开始	44
3.2	快速集成第三方的 Starter.....	53
3.3	降低开发复杂度之面向切面.....	53
3.3.1	前置通知	55
3.3.2	后置返回通知	57
3.3.3	后置异常通知	58
3.3.4	后置最终通知	58
3.3.5	环绕通知	59
3.3.6	AOP 总结	60
3.4	并不复杂的持久化	60
3.4.1	单数据源	61
3.4.2	多数据源	63
3.4.3	JOOQ	65
3.4.4	事务处理	69
3.4.5	整合 Redis	72

3.4.6 整合队列	76
3.4.7 操作 MongoDB	83
3.5 Web 开发	85
3.6 懒人的接口文档管理	89
3.7 优化的调度	92
3.8 健康是永恒的主题	94
3.9 强强联合之整合 Dubbo	96
3.10 小结	101
第 4 章 微服务之 Docker	102
4.1 Docker 原理	104
4.2 更轻量级的虚拟化	105
4.3 三个概念理解 Docker	107
4.3.1 镜像 (Image)	108
4.3.2 容器 (Container)	110
4.3.3 仓库 (Repository)	113
4.4 Dockerfile 定制一切	113
4.4.1 Dockerfile 语法	113
4.4.2 Dockerfile 命令	114
4.4.3 Dockerfile 构建过程	117
4.4.4 构建 Java 环境	118
4.4.5 Dockerfile 小结	120
4.5 Docker 网络	121
4.5.1 网络模式	121
4.5.2 link	121
4.5.3 跨主机访问	122
4.6 Docker 数据卷	122
4.6.1 数据卷	122
4.6.2 数据卷容器	124
4.7 Spring Boot 与 Docker	124
4.8 搭建自己的镜像仓库	128
4.8.1 安装和启动	128

4.8.2 使用	132
4.9 Kubernetes	133
4.10 私有云整体架构	136
4.11 小结	137
第 5 章 微服务之 Spring Cloud	139
5.1 注册中心	142
5.1.1 常用的注册中心	143
5.1.2 Eureka 介绍	144
5.1.3 服务发现	145
5.1.4 简单注册	147
5.2 负载均衡	152
5.2.1 Spring Cloud 的负载实现	154
5.2.2 Ribbon	155
5.2.3 Feign	158
5.2.4 加入 core	161
5.3 微服务容错 (Hystrix)	164
5.3.1 雪崩的形成	164
5.3.2 应对方案	164
5.3.3 降级和熔断	165
5.3.4 Hystrix	166
5.3.5 集中监控	170
5.4 分布式配置中心	172
5.5 API 网关	177
5.5.1 为什么需要网关	178
5.5.2 Zuul	179
5.6 消息总线 (Spring Cloud Bus)	184
5.7 小结	186
第 6 章 微服务之 Spring Cloud 其他框架	187
6.1 Spring Cloud Consul	188
6.2 Spring Cloud ZooKeeper	190

6.3	Spring Cloud archaius.....	192
6.4	Spring Cloud Task.....	193
6.5	Spring Cloud Security.....	194
6.5.1	HTTP Basic Authentication	195
6.5.2	JWT	196
6.5.3	OAuth 2	203
6.5.4	Spring Cloud Security.....	204
6.6	Spring Cloud Sleuth.....	205
6.6.1	服务端	206
6.6.2	客户端	207
6.7	Spring Cloud Stream.....	208
6.8	Spring Cloud Data Flow	211
6.9	小结	212
第 7 章	微服务之自动化测试与质量管理.....	213
7.1	微服务测试	214
7.2	单元测试	216
7.2.1	单元测试及覆盖率评估	216
7.2.2	JUnit	217
7.2.3	Spring Boot 单元测试	218
7.2.4	Mockito.....	220
7.3	API 测试.....	222
7.3.1	Jmeter.....	224
7.3.2	压力测试	225
7.4	A/B 测试.....	227
7.5	冒烟和回归测试	228
7.6	静态代码分析	229
7.6.1	Checkstyle	230
7.6.2	FindBugs.....	233
7.6.3	PMD	234
7.7	SonarQube 质量监控	237
7.7.1	为什么使用	237

7.7.2	安装和使用	238
7.7.3	安装插件	240
7.7.4	运行流程	240
7.8	小结	241
第 8 章	微服务之 JHipster	242
8.1	JHipster 技术列表	243
8.1.1	客户端选项	243
8.1.2	服务端选项	245
8.1.3	部署选项	249
8.2	Angular 简介	250
8.3	快速开始 JHipster	251
8.3.1	安装	251
8.3.2	使用	252
8.3.3	构建单体应用	253
8.3.4	Entity sub-generator	255
8.3.5	开发和运行	258
8.3.6	插件安装	260
8.4	目录结构	260
8.5	构建微服务应用	261
8.5.1	注册中心	261
8.5.2	创建微服务网关	263
8.5.3	Traefik	266
8.5.4	JHipster UAA	266
8.5.5	构建微服务应用	269
8.6	基础配置	271
8.6.1	JHipster 属性配置	271
8.6.2	作为 Maven 项目	274
8.6.3	数据库	274
8.6.4	DTO	275
8.6.5	分页	276
8.6.6	文档	277

8.7 小结	281
第 9 章 微服务之自动化部署	282
9.1 私有仓库搭建	283
9.1.1 Nexus 介绍	283
9.1.2 安装与配置	284
9.1.3 在项目中使用	285
9.2 Ansible	287
9.3 持续集成	289
9.3.1 持续集成流程	290
9.3.2 Jenkins 介绍与安装	291
9.3.3 Maven 介绍	293
9.3.4 Jenkins 系统设置	294
9.3.5 集成 Sonar	295
9.3.6 构建工程	297
9.3.7 配置测试	299
9.4 灰度发布	299
9.5 小结	302
第 10 章 微服务之日志收集与监控	303
10.1 ELK 搜集与分析	305
10.1.1 工作流程	306
10.1.2 日志格式	306
10.1.3 平台搭建	307
10.2 系统监控	310
10.2.1 监控策略和监控对象	310
10.2.2 进程监控	311
10.2.3 数据波动监控	312
10.2.4 常用监控命令	312
10.3 运维监控	316
10.3.1 Zabbix	316
10.3.2 Open-Falcon	321

10.4	APM 监控.....	323
10.4.1	Pinpoint.....	323
10.4.2	SkyWalking.....	325
10.4.3	Zipkin.....	326
10.4.4	CAT.....	328
10.5	Pinpoint 的安装与使用.....	330
10.5.1	Pinpoint 的安装.....	330
10.5.2	Pinpoint 的使用.....	332
10.5.3	Pinpoint 实现邮件告警.....	335
10.6	小结.....	338
第 11 章	完整示例.....	339
11.1	安装 Lombok.....	340
11.2	PiggyMetrics.....	341
11.3	整体架构.....	342
11.3.1	配置 Spring Cloud Config.....	343
11.3.2	授权服务.....	344
11.3.3	API 网关.....	345
11.3.4	服务发现.....	345
11.3.5	负载均衡器、断路器和 HTTP 客户端.....	346
11.3.6	监控仪表盘.....	347
11.3.7	日志分析.....	348
11.4	安装和运行.....	348
11.4.1	配置 Maven 并导入工程.....	348
11.4.2	安装.....	350
11.4.3	使用.....	352
11.4.4	如何变成自己的项目.....	354
11.5	小结.....	355
第 12 章	微服务核心功能推荐.....	356
12.1	工作流引擎.....	357
12.1.1	Activiti.....	357

12.1.2	UFLO.....	358
12.2	规则引擎	360
12.2.1	Drools	360
12.2.2	URule.....	361
12.3	调度系统	362
12.4	消息推送	365
12.5	网关中间件	368
12.5.1	Orange	368
12.5.2	Kong	369
12.5.3	Zuul.....	369
12.6	分库分表中间件	370
12.6.1	Sharding-JDBC.....	370
12.6.2	MyCat	373
12.7	报表引擎	374
12.8	数据处理	375
12.8.1	Spring Batch	376
12.8.2	Kettle	378
12.9	并发编程	379
12.10	分布式配置	380
12.10.1	Disconf.....	380
12.10.2	Apollo	381
12.11	CAS.....	383
12.12	WebFlux.....	384
12.13	小结	388

1 chapter

第1章 微服务概述

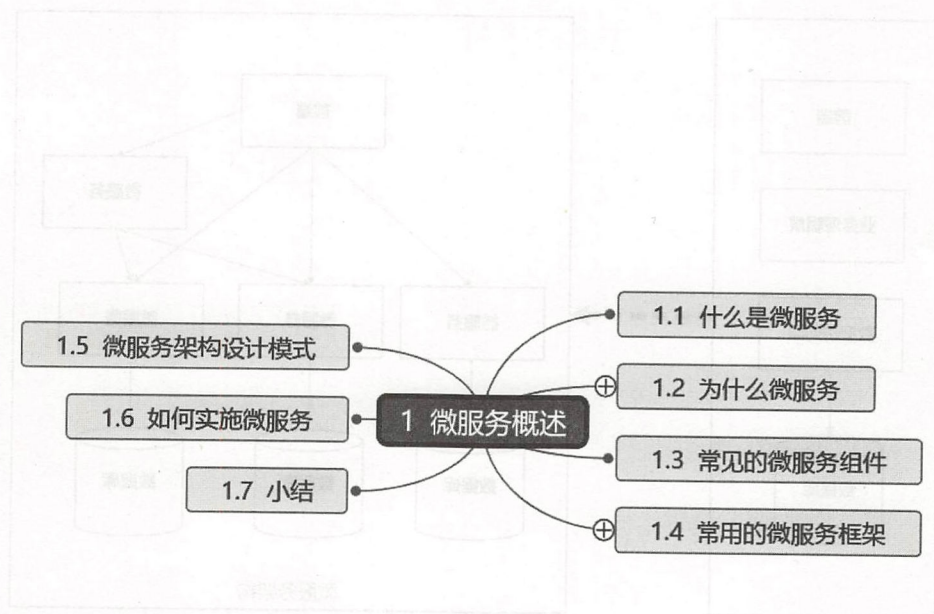


图 1-1 微服务概述

如何构建不同的微服务，将每个微服务部署在不同的服务器上，通过不同的接口进行通信，从而实现微服务的分布式部署。在微服务架构中，每个微服务都是一个独立的单元，可以独立开发、部署和运行。微服务架构的优势在于可以灵活地扩展和升级系统，同时可以降低系统的复杂度和维护成本。在微服务架构中，每个微服务都是一个独立的单元，可以独立开发、部署和运行。微服务架构的优势在于可以灵活地扩展和升级系统，同时可以降低系统的复杂度和维护成本。

1.1 什么是微服务

随着各行各业的快速发展，业务规模的不断扩大，不可避免地造成原有架构不能够适应快速增长和变化。这时，微服务就进入大家的视野，其实在微服务之前，很多公司已经做过服务化的改造，并且取得了一定的成果，但是对于整体流程的标准化还有一定有差距。那么，什么是微服务呢？

准确地说，微服务是一种软件架构模式，将大型系统或者复杂的应用分割成多个服务的架构，服务之间互相协调、互相配合，为用户提供最终价值。每个服务都有独立的生命周期，可以单独维护和部署，各个业务模块之间是松耦合的，比传统的应用程序更有效地利用了计算资源，应用的扩展更加灵活，能够通过扩展组件来处理功能瓶颈问题。这样一来，开发人员只需要为额外的组件部署计算资源，而不需要部署一个完整的应用程序的全新迭代。

一个微服务的架构如图 1-1 所示，单体应用被拆分成多个微小的服务。

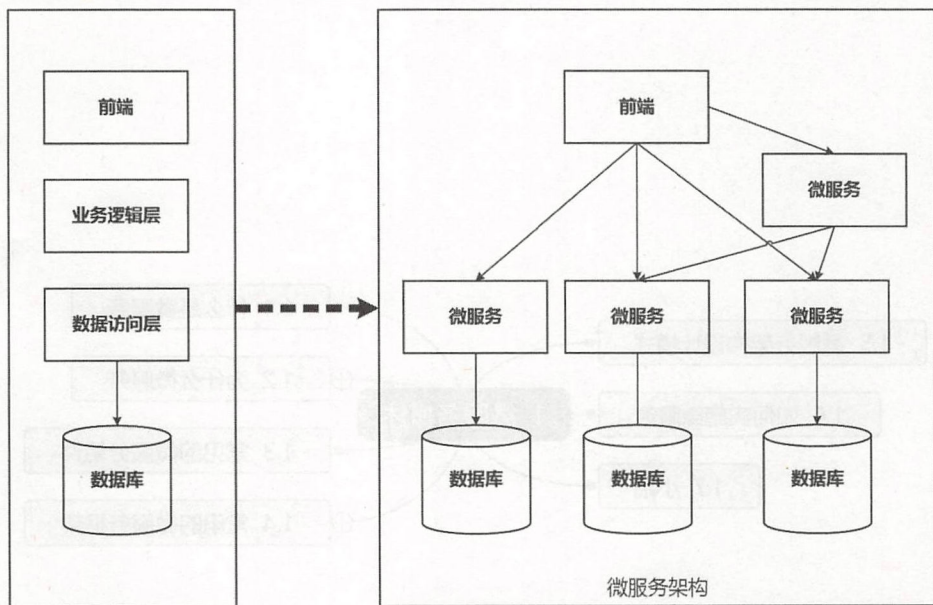


图 1-1 微服务架构

也有人将微服务的开发比喻成搭积木，每个服务都是一个零件，使用这些不同的服务可以搭建出不同的形状。简单地说，微服务架构就是把一个大系统按业务功能分解成多个职责单一的小系统，并利用简单的方法使多个小系统相互协作，组合成一个大系统。

其实这里蕴含着自古以来的真理，就是分而治之，当一件事情大到不能处理的时候，就使用一定的切分方法，将其变成很多微小的类，然后分门别类地进行处理，以达到最好的效果，

最终实现 $1+1>2$ 的功效。

微服务的优点和缺点（或者说挑战）一样明显。

优点

- 开发简单：每个服务完成独立的功能；
- 技术栈灵活：可以选择不同的语言完成不同的服务，发挥各种语言的最大优势；
- 服务独立无依赖：每个服务都可以单独部署，一个服务出现问题不会导致整个系统瘫痪；
- 独立按需扩展：以应对高并发与大流量；
- 可用性高：当其中一个点出现问题时，能够及时切换，不影响业务的正常运行；
- 复杂应用解耦为小而众的服务：拆分可以基于一定的原则，将耗时的应用解耦；
- 各服务精而专：也就是我们常说的专人干专事，映射到微服务中就是专服务干专事；
- 服务间通信通过 API 完成：选择轻量的 API 通信。

缺点（挑战）

- 多服务运维难度：服务的增加意味着运维的困难，如何有效地管理是一个挑战；
- 系统部署依赖：当业务复杂时，系统之间的耦合关系高度耦合，如何高效部署是一个挑战；
- 服务间通信成本：包括网络延迟、接口不可用性等，保证服务的高可用性是一个挑战；
- 数据一致性：各个服务间如何有效地共享数据，确保相应服务的数据需求一致性是一个挑战；
- 系统集成测试：拆分后，原本需要测试的内容成倍地增加，如何高效地降低测试成本是一个挑战；
- 重复工作：服务拆分之后，由于信息的不对称导致的重复性工作，如何有效抽象是一个挑战；
- 性能监控：原本只需要一个监控的部分，现在需要分开监控，如何快速定位问题是一个挑战；
- 沟通成本的成倍增加：服务拆分后，各个服务由单独的人来维护，如何高效地沟通是一个挑战。

1.2 为什么微服务

从一般的平台遇到的问题说起：

- 服务配置复杂。基础服务多，服务的资源配置复杂。传统方式管理服务复杂。
- 服务之间调用复杂。检索服务、用户中心服务等，服务之间的调用复杂，依赖多。
- 服务监控难度大。服务比较多，机器部署复杂，服务存活监控、业务是否正常监控尤为重要。
- 服务化测试问题。服务依赖性比较大，测试一个小的功能，周边服务也需要启动。

那么微服务的架构有什么优势，大家为什么都怀着极高的热情来应对微服务呢？

1.2.1 区别

首先看一下两者的区别，单体架构和分布式微服务架构的优点和缺点如表 1-1 所示。

表 1-1 单体架构和分布式微服务架构对比

	传统单体架构	分布式微服务架构
新功能开发	需要时间	容易开发和实现
部署	不经常且容易部署	经常发布，部署复杂
隔离性	故障影响范围大	故障影响范围小
架构设计	难度小	难度级数增加
系统性能	响应时间快，吞吐量小	响应时间慢，吞吐量大
系统运维	运维简单	运维复杂
技术	技术单一且封闭	技术多样且开放
测试和查错	简单	复杂
系统扩展性	扩展性很差	扩展性很好
系统管理	重点在于开发成本	重点在于服务治理和调度

从上面的表格我们可以看到，分布式系统虽然有一些优势，但也存在一些问题：

- 架构设计变得复杂（尤其是其中的分布式事务）；
- 部署单个服务会比较快，但一次部署多个服务会变得复杂；
- 系统的吞吐量会变大，但是响应时间会变长；
- 运维复杂度会因为服务变多而变得很复杂；
- 架构复杂导致学习曲线变大；
- 测试和查错的复杂度增大；
- 技术很多样，带来维护和运维的复杂度提升；
- 管理分布式系统中的服务和调度变得困难且复杂。

也就是说，分布式系统架构的难点在于系统的设计、管理和运维。

接下来我们看一下架构的演变过程。

1.2.2 从单体应用说起

图 1-2 是我们非常熟悉的单体应用，或者说是非常传统的应用架构。

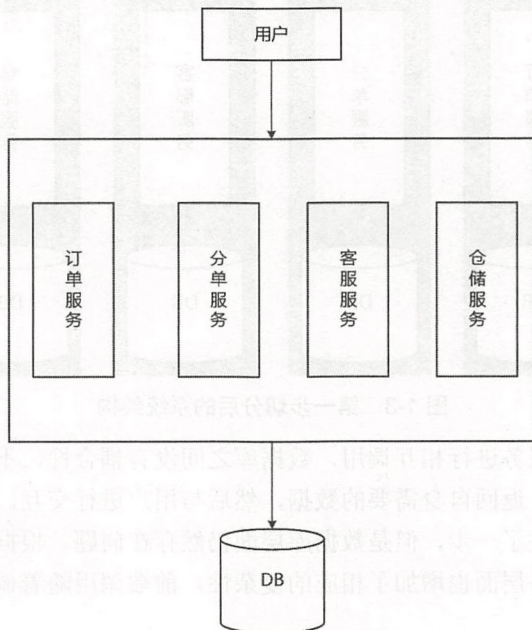


图 1-2 传统单体应用

从图 1-2 中可以看到，应用通过浏览器进行访问，当访问到订单服务时，分单服务提供相应的功能，然后去访问应用的数据层，数据层负责对数据进行解析。这是一个极其典型的单体应用结构，这种结构所带来的问题显而易见，数据库存在单点，所有服务都耦合在一个应用中，当其中一个服务出现问题时，整个工程都需要重新发布，从而导致整体业务不能提供响应。这种结构在小项目中是没有什么问题的，而且操作起来非常灵活，但当业务量爆增的时候，就无法灵活应对了。

1.2.3 第一步切分

单体架构无法满足业务增长的需求，需要对数据库进行进一步的切分，以提高扩展性，图 1-3 就是一个切分后的架构图。

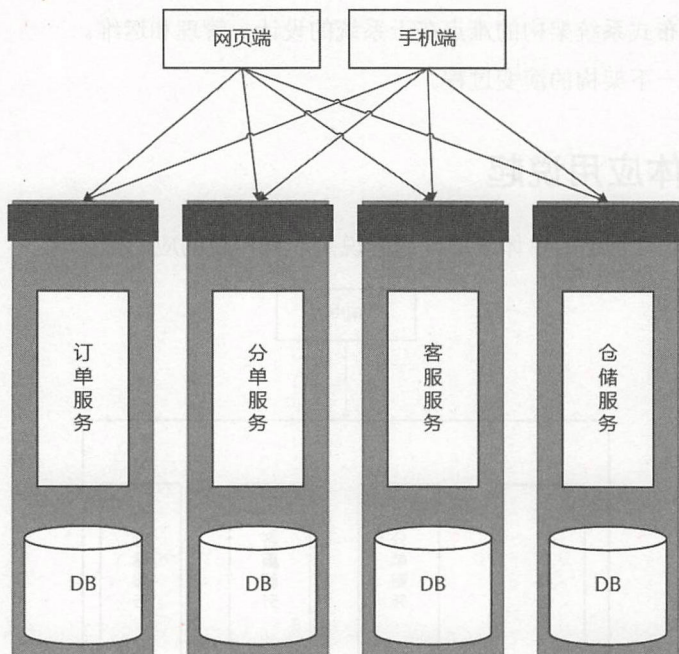


图 1-3 第一步切分后的系统架构

业务通过进程间的服务进行相互调用，数据库之间没有耦合性，不会存在单点故障。前端只需要调用相应的服务，返回自身需要的数据，然后与用户进行交互。可以看到，这种方式比传统的单体应用已经前进了一步，但是数据库层面仍然存在问题。根据数据量需要评估是否使用读写分离的设计，服务层面也增加了相应的复杂性，前端调用随着调用接口数量的增加也急需治理。

1.2.4 服务化所带来的问题

随着服务的拆分，新的问题随之而来。客户端如何访问这些服务？这些服务的调用情况？切分是否合理？是否安全？如果受到攻击应该如何应对？是否可以使用限流或者降级的方式来及时解决？

应对这些问题，API 网关是一个不错的解决方案。当有新的设备需要调用这些接口时，可以复用原有接口，不需要进行二次开发。接口的维护也会更有条理性，对于访问次数、安全等问题，都可以在这一层进行解决，如图 1-4 所示。

解决了应用前端访问的问题后，服务之间的相互调用也是一个问题，如果整个系统内部调用关系混乱，就会带来非常多的不必要的问题。所以约定好服务之间的通信方式是非常有必要的。不管是开源还是公司内部研发，都有非常多的解决方案。最简单的方式是通过 RPC 的调用，

传输 JSON 或者 XML，双方定义好协议格式，通过报文的方式进行数据传输。

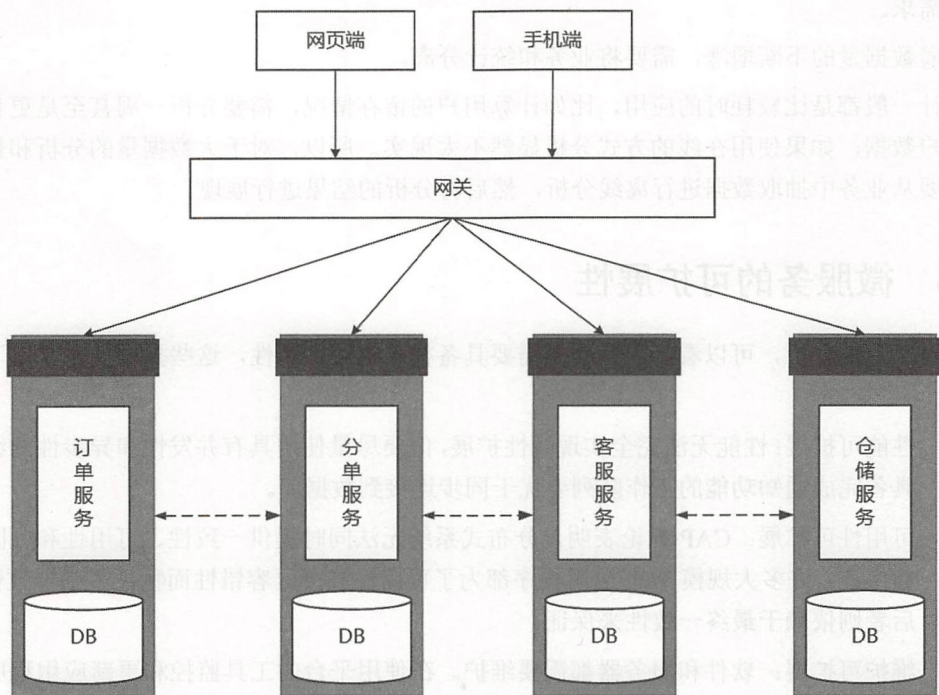


图 1-4 加入网关

当多种服务需要互相调用时，服务的数量会急剧地增加，服务的治理就成为新的问题，另外还有不同服务的版本问题。如果不能通过一个单独的注册地址，像书的目录一样来管理整个服务结构，服务的调用就显示非常混乱。

一般的分布式服务都有一个注册中心，例如，Dubbo 是基于 ZooKeeper 进行的二次开发，自身提供管理控制台，可以对服务进行注册和查找，Spring Cloud 有 Eureka，还有 etcd、Consul 等可供选择。

经过上面的处理，整体上来看应用已经达到了一个非常好的状态，但是每个应用服务仍然存在着单点问题，当一个服务出现问题时，有可能导致连锁的反应，使整个系统瘫痪。这时需要除监控告警外的一种容错机制来保障整体服务的可运行性。

通过网关层的反向代理来实现高可用是一个不错的解决方案，访问层无须了解整个体系有多少应用提供，只需要关心服务是否能够提供服务，并且对必要的接口进行监测即可。当发现接口无法正常提供服务时，提供相应的告警机制，以微信、短信或者邮件的方式通知相关人及时进行处理。

随着服务的切分、业务的扩展，数据量的激增也是一个非常大的问题，如果采用传统的方

案来应对，各种关系型数据库都有瓶颈，把运算量比较大、比较耗资源的运算通过跨库来统计查询的需求。

随着数据量的不断增涨，需要将业务和统计分离。

统计一般都是比较耗时的应用，比如计算用户的留存情况，需要分析一周甚至是更长周期内的用户数据，如果使用在线的方式分析显然不太现实。所以，对于大数据量的分析和数据挖掘，需要从业务中抽取数据进行离线分析，然后将分析的结果进行展现。

1.2.5 微服务的可扩展性

针对以上的分析，可以看到，微服务需要具备极强的可扩展性，这些扩展性包含以下几个方面。

- 性能可扩展：性能无法完全实现线性扩展，但要尽量使用具有并发性和异步性的组件。具备完成通知功能的工作队列要优于同步连接到数据库。
- 可用性可扩展：CAP 理论表明，分布式系统无法同时提供一致性、可用性和分区容错性保证。许多大规模 Web 应用程序都为了可用性和分区容错性而牺牲了强一致性，而后者则依赖于最终一致性来保证。
- 维护可扩展：软件和服务器都需要维护。在使用平台的工具监控和更新应用程序时，要尽可能自动化。
- 成本可扩展：总成本包括开发、维护和运营支出。在设计一个系统时，要在重用现有组件和完全新开发组件之间进行权衡。现有组件很少能完全满足需求，但修改现有组件的成本还是可能低于开发一个完全不同的方案的。另外，使用符合行业标准的技术使组织更容易聘到专家，而发布独有的开源方案则可能帮助组织从社区中挖掘人才。

1.2.6 微服务与 SOA 的区别

面向服务的架构（SOA）是一个组件模型，它将应用程序的不同功能单元（称为服务）通过这些服务之间定义良好的接口和契约联系起来。接口是采用中立的方式进行定义的，它应该独立于实现服务的硬件平台、操作系统和编程语言。这使得构建在各种各样系统中的服务可以以一种统一和通用的方式进行交互。

都是做服务化，那么微服务与 SOA 的异同有哪些呢？

相同点

- 需要 Registry，实现动态的服务注册发现机制。

- 需要考虑分布式下面的事务一致性，CAP 原则下，两段式提交不能保证性能，事务补偿机制需要考虑。
- 同步调用还是异步消息传递，如何保证消息可靠性？SOA 由 ESB 来集成所有的消息。
- 都需要统一的 Gateway 来汇聚、编排接口，实现统一认证机制，对外提供 APP 使用的 RESTful 接口。
- 同样要关注如何再分布式下定位系统问题，如何做日志跟踪？就像电信领域做了十几年的信令跟踪的功能。

差异点

- 是持续集成、持续部署？对于 CI、CD（持续集成、持续部署），这本身和敏捷、DevOps 是交织在一起的，所以更倾向于软件工程的领域而不是微服务技术本身。
- 使用不同的通信协议是不是区别？微服务的标杆通信协议是 RESTful，而传统的 SOA 一般是 SOAP，不过目前来说采用轻量级的 RPC 框架（Dubbo、Thrift、gRPC）非常多，在 Spring Cloud 中也有 Feign 框架将标准 RESTful 转为代码的 API 这种仿 RPC 的行为，这些通信协议不应该是区分微服务架构和 SOA 的核心差别。
- 是流行的基于容器的框架还是虚拟机为主？Docker 虚拟机和物理机都是架构实现的一种方式，不是核心区别。

SOA 和微服务的一个主要不同点就是自动化程度上的不同。大部分的 SOA 实现只达到服务级别的抽象，而微服务达到了对实现和运行环境的抽象级别。

在一个规范的微服务中，每个微服务应该被构建成胖 JAR（fat JAR），其中内置了所有的依赖，然后作为一个单独的 Java 进程存在。

1.3 常见的微服务组件

既然谈到了微服务架构，下面介绍通用的微服务都包括哪些组件。

➤ 服务注册

服务注册是一个记录当前可用的微服务实例的网络信息数据库，是服务发现机制的主要核心，服务注册表包含查询 API、管理 API，使用查询 API 获得可用服务的实例，使用管理 API 实现注册、注销。

➤ 服务发现

服务调用方从服务注册中心找到自己需要调用的服务的地址。可以选择客户端服务发现，也可以选择服务端服务发现。

➤ 负载均衡

服务提供方一般以多实例的形式提供服务，负载均衡功能能够让服务调用方连接到合适的服务节点，并且节点选择的工作对服务调用方来说是透明的。可以选择服务端的负载均衡，也可以选择客户端的负载均衡。

➤ 服务网关

服务网关是服务调用的唯一入口，可以在这个组件是实现用户鉴权、动态路由、灰度发布、A/B 测试、负载限流等功能。根据公司流量规模的大小网关可以是一个，也可以是多个。

➤ 配置中心

将本地化的配置信息（Properties、XML、YAML 等）注册到配置中心，实现程序包在开发、测试、生产环境的无差别性，方便程序包的迁移。配置部分可以单独使用高可用的分布式配置中心，确保一个配置服务出现问题时，其他服务也能够提供配置服务。

➤ API 管理

以方便的形式编写及更新 API 文档，并以方便的形式供调用者查看和测试。通常需要加入版本控制的概念，以确保服务的不同版本在升级过程中都能够提供服务。

➤ 集成框架

微服务组件都以职责单一的程序包对外提供服务，集成框架以配置的形式将所有微服务组件集成到统一的界面框架下，让用户能够在统一的界面中使用系统。

➤ 分布式事务

对于重要的业务，需要通过分布式事务技术（TCC、高可用消息服务、最大努力通知）保证数据的一致性。根据业务的不同，适当地牺牲一些数据的一致性要求，确保数据的最终一致性。

➤ 调用链

记录完成一个业务逻辑时调用到的微服务，并将这种串行或并行的调用关系展示出来。在系统出错时，可以方便地找到出错点。同时统计各个服务的调用次数，确保比较热的服务能够被分配更多的资源。

➤ 支撑平台

系统微服务化后，系统变得更加碎片化，系统的部署、运维、监控等都比单体架构更加复杂，那么就需要将大部分的工作自动化。现在，可以通过 Docker 等工具来中和这些微服务架构带来的弊端。例如，持续集成、蓝绿发布、健康检查、性能健康等。可以这么说，如果没有合适的支撑平台或工具，就不要使用微服务架构。

1.4 常用的微服务框架

各种语言都有自己的微服务框架，主要包括：

- 主流微服务框架（Spring Cloud、Dubbo）；
- 新锐微服务框架（Istio）。

下面看一下常用的微服务框架体系。

1.4.1 Spring 顶级项目

首先看一下 Spring 的顶级项目都包括哪些。

- **Spring I/O platform**: 用于系统部署，是可集成的、构建现代化应用的版本平台，具体来说当你使用 Maven Dependency 引入 Spring jar 包时它就在工作了。
- **Spring Boot**: 旨在简化创建产品级的 Spring 应用和服务，简化了配置文件，使用嵌入式 Web 服务器，含有诸多开箱即用的微服务功能，可以和 Spring Cloud 联合部署。
- **Spring Framework**: 即通常所说的 Spring 框架，是一个开源的 Java/Java EE 全功能栈应用程序框架，其他 Spring 项目如 Spring Boot 也依赖于此框架。
- **Spring Cloud**: 微服务工具包，为开发者提供了在分布式系统的配置管理、服务发现、断路器、智能路由、微代理、控制总线等开发工具包。
- **Spring XD**: 一种运行时环境（服务器软件，非开发框架），组合 Spring 技术，如 Spring batch、Spring Boot、Spring Data，采集大数据并处理。
- **Spring Data**: 一个数据访问及操作的工具包，封装了很多种数据及数据库的访问相关技术，包括 JDBC、Redis、MongoDB、Neo4j 等。
- **Spring Batch**: 批处理框架，或者说是批量任务执行管理器，功能包括任务调度、日志记录/跟踪等。
- **Spring Security**: 一个能够为基于 Spring 的企业应用系统提供声明式的安全访问控制解决方案的安全框架。
- **Spring Integration**: 面向企业应用集成（EAI/ESB）的编程框架，支持的通信方式包括 HTTP、FTP、TCP/UDP、JMS、RabbitMQ、E-mail 等。
- **Spring Social**: 一组工具包，一组连接社交服务 API，如 Twitter、Facebook、LinkedIn、GitHub 等，有几十个。
- **Spring AMQP**: 消息队列操作的工具包，主要是封装了 RabbitMQ 的操作。

- Spring HATEOAS: 一个用于支持实现超文本驱动的 REST Web 服务的开发库。
- Spring Mobile: Spring MVC 的扩展, 用来简化手机上的 Web 应用开发。
- Spring for Android: Spring 框架的一个扩展, 其主要目的在于简化 Android 本地应用的开发, 提供 RestTemplate 来访问 REST 服务。
- Spring Web Flow: 目标是成为管理 Web 应用页面流程的最佳方案, 将页面跳转流程单独管理, 并可配置。
- Spring LDAP: 一个用于操作 LDAP 的 Java 工具包, 基于 Spring 的 JdbcTemplate 模式, 简化 LDAP 访问。
- Spring Session: Session 管理的开发工具包, 让你可以把 Session 保存到 Redis 等, 进行集群化 Session 管理。
- Spring Web Services: 基于 Spring 的 Web 服务框架, 提供 SOAP 服务开发, 允许通过多种方式创建 Web 服务。
- Spring Shell: 提供交互式的 Shell, 可以使用简单的基于 Spring 的编程模型来开发命令, 比如 Spring Roo 命令。
- Spring Roo: 一种 Spring 开发的辅助工具, 使用命令行操作来生成自动化项目, 操作非常类似于 Rails。
- Spring Scala: Scala 语言编程提供的 Spring 框架的封装 (新的编程语言, Java 平台的 Scala 于 2003 年年底/2004 年年初发布)。
- Spring BlazeDS Integration: 一个开发 RIA 工具包, 可以集成 Adobe Flex、BlazeDS、Spring 及 Java 技术创建 RIA。
- Spring Loaded: 用于实现 Java 程序和 Web 应用的热部署的开源工具。
- Spring REST Shell: 可以调用 REST 服务的命令行工具, 使用命令行操作 REST 服务。

1.4.2 背景

对于使用 Java 技术更多的团队来说, Dubbo、Spring Cloud 的知名度相对较高, 而且在业界的影响力非常高。那么我们就来讨论一下这两种框架, 做一个综合的评比。

➤ Dubbo

Dubbo 是阿里出品的服务化组件, 应用于多个部门, 后来开源之后, 也衍生出像 Dubbox 这样的框架, 对其进行了进一步的增强。Dubbo 是一个分布式服务框架, 是国内互联网公司开源做得比较不错的微服务化治理框架, 致力于提供高性能、透明化的 RPC 远程服务调用方案和

SOA 服务治理方案，如图 1-5 所示。目前 Dubbo 已经正式进入 Apache 孵化器。

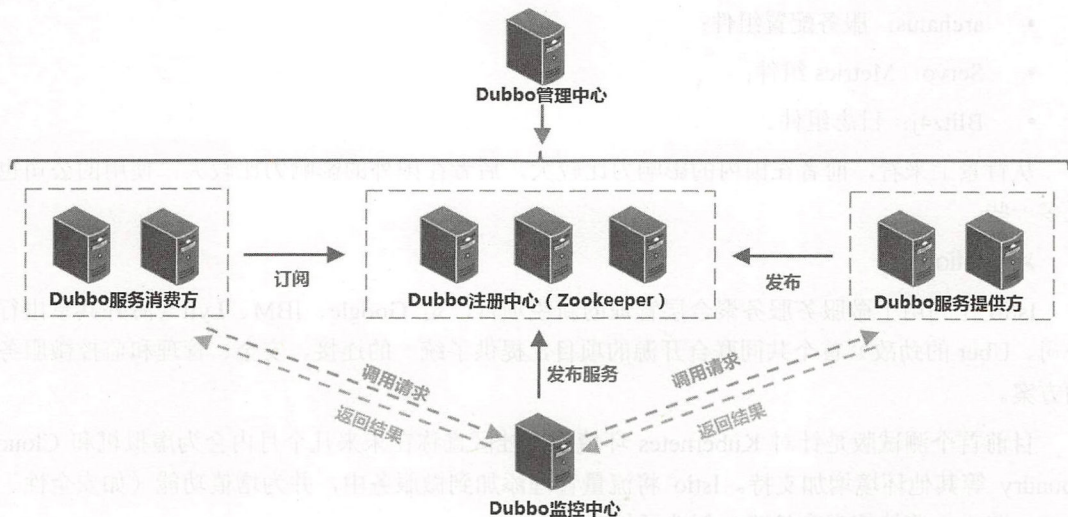


图 1-5 Dubbo 分布式服务

其核心部分包含：

- 远程通信——提供对多种基于长连接的 NIO 框架抽象封装，包括多种线程模型、序列化，以及“请求—响应”模式的信息交换方式。
- 集群容错——提供基于接口方法的透明远程过程调用，包括多协议支持，以及软负载均衡、失败容错、地址路由、动态配置等集群支持。
- 自动发现——基于注册中心目录服务，使服务消费方能动态的查找服务提供方，使地址透明，使服务提供方可以平滑增加或减少机器。

➤ Spring Cloud

从命名就可以知道，Spring Cloud 是 Spring Source 的产物，可以说 Spring 的出现完全改变了企业级开发，Spring Cloud 除了具有 Spring 社区的强大背景，还有 Netflix 强大的后盾与技术输出。Netflix 作为一家成功实践微服务架构的互联网公司，在几年前就把几乎整个微服务框架栈开源贡献给了社区，这些框架开源的整套微服务架构套件是 Spring Cloud 的核心。

- Eureka：服务注册发现框架；
- Zuul：服务网关；
- Karyon：服务端框架；
- Ribbon：客户端框架；

- Hystrix: 服务容错组件;
- archaius: 服务配置组件;
- Servo: Metrics 组件;
- Blitz4j: 日志组件。

从背景上来看,前者在国内的影响力比较大,后者在国外的影响力比较大,使用的公司也更多一些。

➤ Istio

Istio 作为用于微服务服务聚合层管理的新锐项目,是 Google、IBM、Lyft (海外共享出行公司、Uber 的劲敌) 首个共同联合开源的项目,提供了统一的连接、安全、管理和监控微服务的方案。

目前首个测试版是针对 Kubernetes 环境的,社区宣称在未来几个月内会为虚拟机和 Cloud Foundry 等其他环境增加支持。Istio 将流量管理添加到微服务中,并为增值功能(如安全性、监控、路由、连接管理和策略)创造了基础。

- HTTP、GRPC 和 TCP 网络流量的自动负载均衡;
- 提供了丰富的路由规则,实现细粒度的网络流量行为控制;
- 流量加密、服务间认证,以及强身份声明;
- 全范围 (Fleet-wide) 的策略执行;
- 深度遥测和报告。

1.4.3 社区活跃度

选择一个开源框架,社区的活跃度是我们极为关注的一个要点。社区越活跃,解决问题的速度越快,框架也会越来越完善,不然当我们碰到问题时,就不得不自己解决。

图 1-6 是 Dubbo 目前的情况,在 2017 年,阿里已经宣布有团队重新维护 Dubbo 项目,可以看到 Dubbo 项目已经重新开始修复里边的一些 Bug,并且进入不断地完善过程中了。目前 Dubbo 在 GitHub 上有超过 16000 个 star 和超过 12000 的 fork 数,是国内影响力最大的开源项目之一。

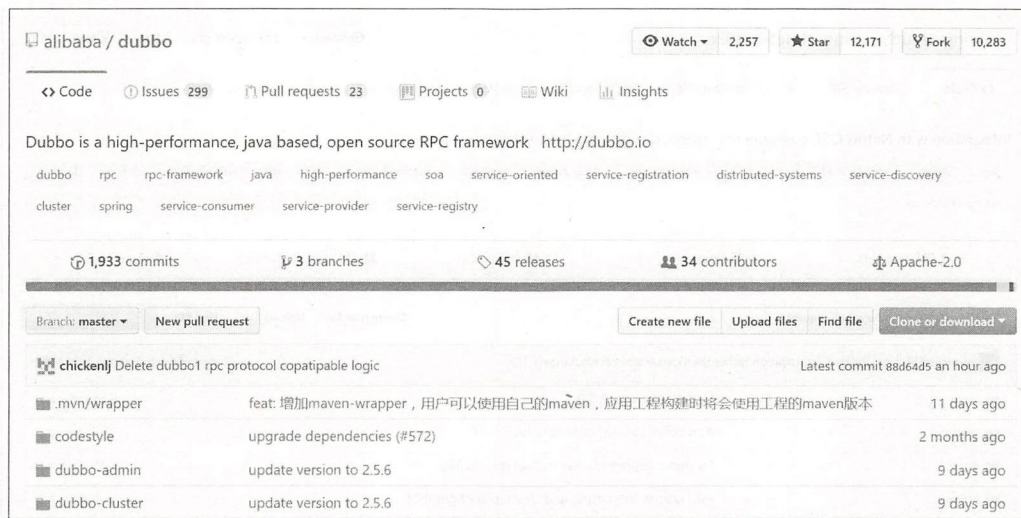


图 1-6 Dubbo 更新情况

反倒是继承而来的 Dubbox，已经有很长时间没有进行维护了，如图 1-7 所示。

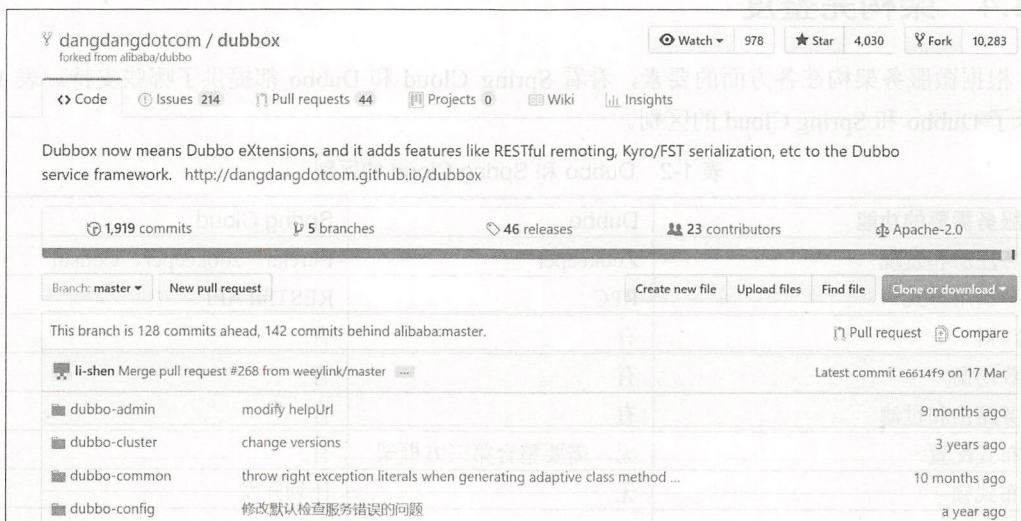


图 1-7 Dubbox 更新情况

Spring Cloud 沿袭 Spring 一贯的风格，更新非常及时，响应的速度也非常快，社区非常活跃，如图 1-8 所示。

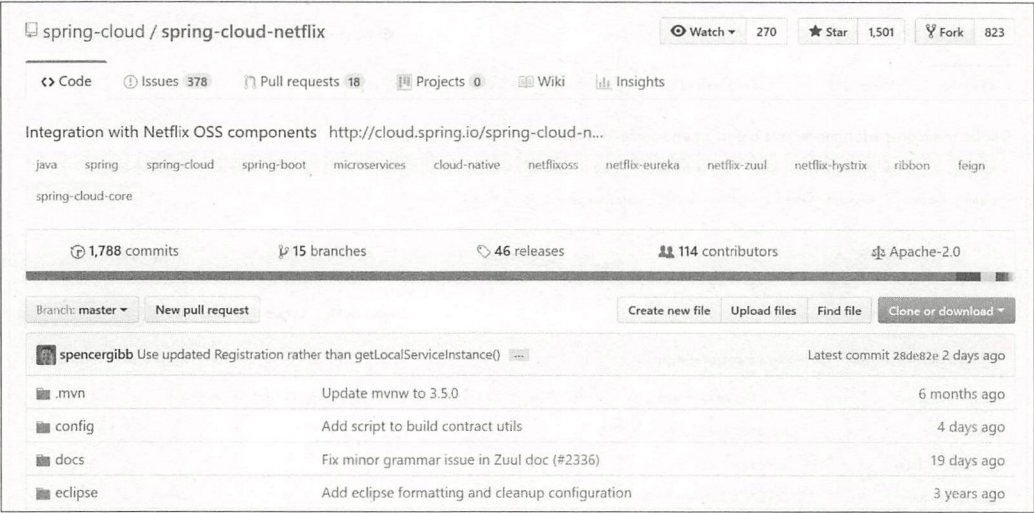


图 1-8 Spring Cloud 更新情况

1.4.4 架构完整度

根据微服务架构在各方面的要素，看看 Spring Cloud 和 Dubbo 都提供了哪些支持。表 1-2 展示了 Dubbo 和 Spring Cloud 的区别。

表 1-2 Dubbo 和 Spring Cloud 的区别

微服务需要的功能	Dubbo	Spring Cloud
服务注册和发现	Zookeeper	Eureka、zookeeper、Consul
服务调用方式	RPC	RESTful API
断路器	有	有
负载均衡	有	有
服务路由和过滤	有	有
分布式配置	无，需要整合第三方框架	有
分布式锁	无	计划开发
集群选主	无	有
分布式消息	无	有

其实单纯从提供的数量上来比较，有一些不公平，因为 Dubbo 是几年前出来的，极大地改变了人们对于分布式系统的认知。而表 1-2 中写无的部分其实也可以整合其他的框架。所以，对于这一点的评比上来说，选用何种框架取决于团队目前的情况，而不是一下子全部转型，要阶梯式地实现整个应用架构。

1.4.5 总结

使用 Dubbo 构建的微服务架构就像组装计算机，各环节的选择自由度很高，但最终结果很有可能因为一条内存质量不行导致不能开机，总是让人不怎么放心。如果你是一名高手，则这些都不是问题。而 Spring Cloud 就像品牌机，在 Spring Source 的整合下做了大量的兼容性测试，保证了机器拥有更高的稳定性。如果要使用非原装组件外的东西，则需要对其基础有足够的了解。

1.5 微服务架构设计模式

➤ 聚合器微服务设计模式

这是一种最常见也最简单的设计模式，效果如图 1-9 所示。

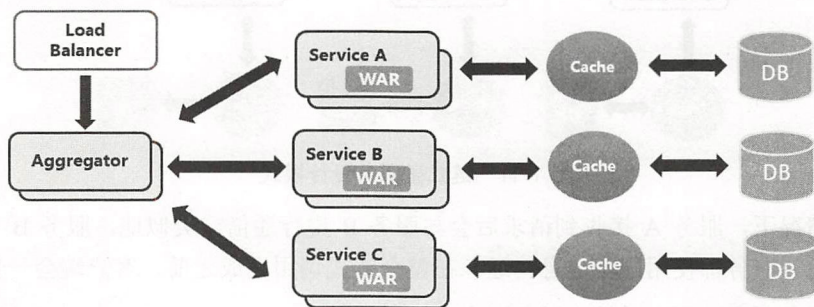


图 1-9 聚合器微服务设计模式

聚合器调用多个服务实现应用程序所需的功能。它可以是一个简单的 Web 页面，将检索到的数据进行处理并展示，也可以是一个更高层次的组合微服务，对检索到的数据增加业务逻辑后进一步发布成一个新的微服务，这符合 DRY 原则。另外，每个服务都有自己的缓存和数据库。如果聚合器是一个组合服务，那么它也有自己的缓存和数据库。

➤ 代理微服务设计模式

这是聚合模式的一个变种，如图 1-10 所示。

在这种情况下，客户端并不聚合数据，但会根据业务需求的差别调用不同的微服务。代理可以仅仅委派请求，也可以进行数据转换工作。每个微服务都有自己独立的缓存和数据库系统，彼此独立。

➤ 链式微服务设计模式

这种模式在接收到请求后会产生一个经过合并的响应，如图 1-11 所示。

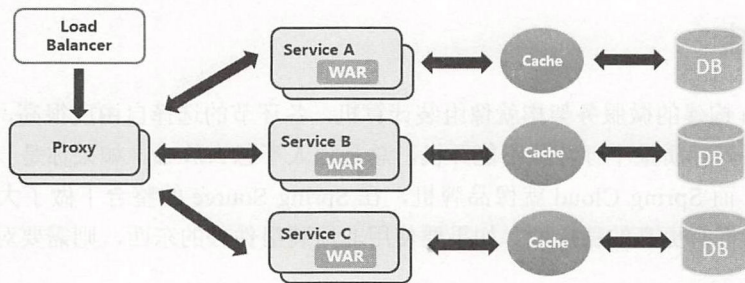


图 1-10 代理微服务设计模式

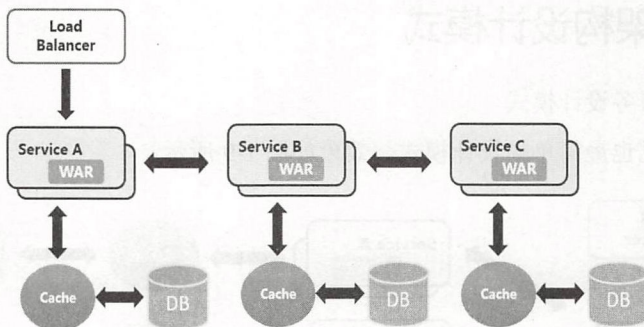


图 1-11 链式微服务设计模式

在这种情况下，服务 A 接收到请求后会与服务 B 进行通信，类似地，服务 B 会同服务 C 进行通信。所有服务都使用同步消息传递。在整个链式调用完成之前，客户端会一直阻塞。

因此，服务调用链不宜过长，以免客户端长时间等待。

➤ 分支微服务设计模式

这种模式是聚合器模式的扩展，允许同时调用两个微服务链，如图 1-12 所示。

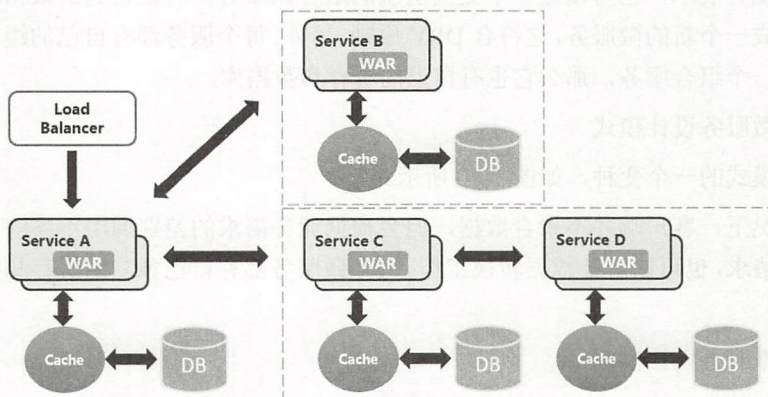


图 1-12 分支微服务设计模式

每个调用链分别调用自己的服务。当某个调用出现问题时，相互之间不会造成影响。

➤ 数据共享微服务设计模式

自治是微服务的设计原则之一，也就是说微服务是全栈式服务。但在重构现有的“单体应用（monolithic application）”时，SQL 数据库反规范化可能会导致数据重复和不一致。

因此，在单体应用到微服务架构的过渡阶段，可以使用这种设计模式，如图 1-13 所示。

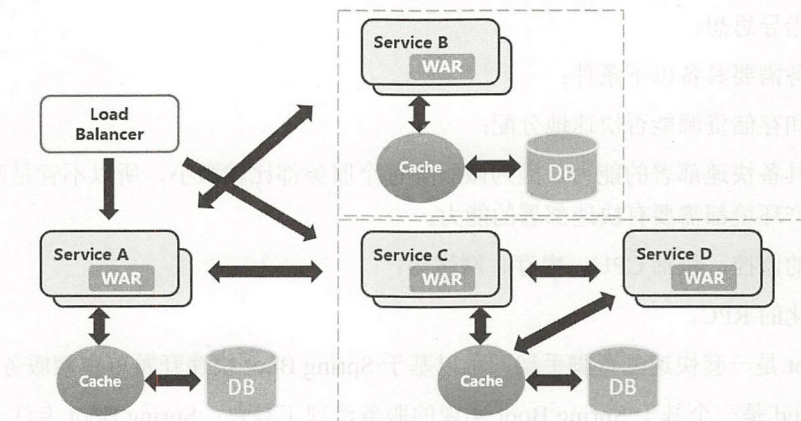


图 1-13 数据共享微服务设计模式

在这种情况下，部分微服务可能会共享缓存和数据库存储。不过，这只有在两个服务之间存在强耦合关系时才可以。对于基于微服务的新建应用程序而言，这是一种反模式。

➤ 异步消息传递微服务设计模式

虽然 REST 设计模式非常流行，但它是同步的，会造成阻塞。因此部分基于微服务的架构可能会选择使用消息队列代替 REST 请求/响应，如图 1-14 所示。

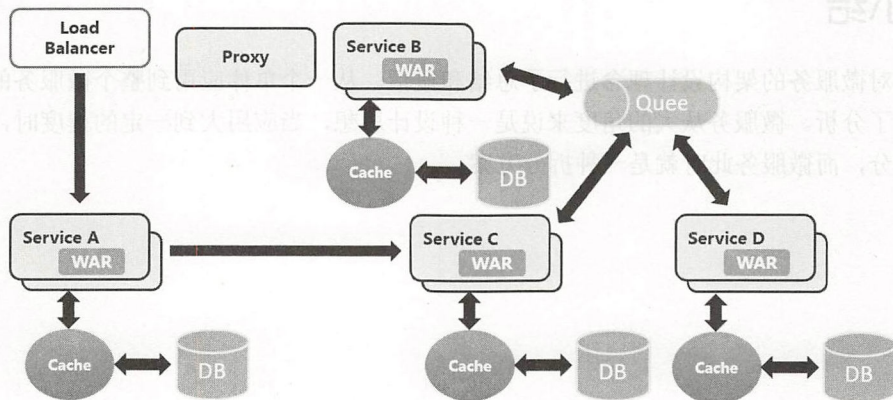


图 1-14 异步消息传递微服务设计模式

各个服务之间通过异步的消息队列进行交互，当服务出现问题时，不会造成阻塞，队列会帮助缓存消息，直到消费服务开始工作。

1.6 如何实施微服务

微服务是一种架构的理念，Martin Fowler 提出了微服务的设计原则，从理论上为具体的技术落地提供了指导思想。

实施微服务需要具备以下条件：

- 计算和存储资源能否快速地分配；
- 是否具备快速部署的能力，因为微服务每个服务都比较微小，所以不管是测试环境还是生产环境都需要有快速部署的能力；
- 基本的监控，包括 CPU、内存、网络等；
- 标准化的 RPC。

Spring Boot 是一套快速配置脚手架，可以基于 Spring Boot 快速开发单个微服务。

Spring Cloud 是一个基于 Spring Boot 实现的服务治理工具包；Spring Boot 专注于快速、方便集成的单个微服务个体；Spring Cloud 关注全局的服务治理框架。

Spring Boot / Cloud 是微服务实践的最佳落地方案。

当然，微服务的设计还对运维提出了更高的要求，如何进行自动构建，如何进行自动发布，应用程序的质量管理，以及遇到峰值时如何通过横向扩展、弹性伸缩，对整个技术团队都提出了更高的要求。

1.7 小结

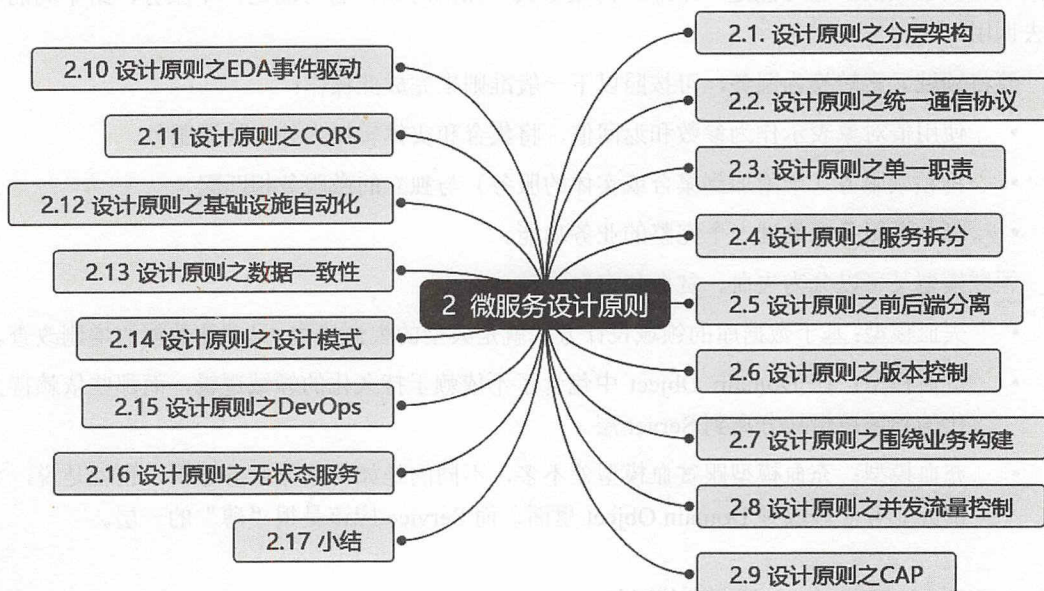
本章对微服务的架构设计理念进行了总结和概括。从一个单体应用到整个微服务的拆分过程都进行了分析。微服务从大的角度来说是一种设计思想，当应用大到一定的程度时，就不得不进行拆分，而微服务此时就是一种拆分方案。





2 chapter

第 2 章 微服务设计原则



领域驱动设计 DDD (Domain Driven Design) 提出了从业务设计到代码实现一致性的要求, 不再对分析模型和实现模型进行区分。也就是说, 从代码的结构中我们可以直接理解业务的设计, 命名得当的话, 非程序人员也可以“读”代码。这与微服务设计中的约定优于配置不谋而合, 如果你熟悉英文, 那么根据包名和类名就可以解读出程序开发者所构建的业务的大概意图。

领域模型包含一些明确定义的类型:

- 实体是一个对象, 它有固定的身份, 具有明确定义的“连续性线索”或生命周期。通常列举的示例是一个 `Person` (一个实体)。大多数系统都需要唯一地跟踪一个 `Person`, 无论姓名、地址或其他属性是否更改。
- 值对象没有明确定义的身份, 而仅由它们的属性定义。它们通常不可变, 所以两个相等的值对象始终保持相等。地址可以是与 `Person` 关联的值对象。
- 集合是一个相关对象集群, 这些对象被看作一个整体。它拥有一个特定实体作为它的根, 并定义了明确的封装边界。它不只是一个列表。
- 服务用于表示不是实体或值对象的自然部分的操作或活动。

领域模型在实现时可大可小, 在业务的早期, 在系统比较小的情况下, 它有可能是一个类。当系统做大了以后, 它可能是一个库。再做更大一点的时候, 它可能是一个服务, 给不同的应用去调用。

要将领域元素转换为服务, 可按照以下一般准则来完成此操作:

- 使用值对象表示作为参数和返回值, 将集合和实体转换为独立的微服务。
- 将领域服务 (未附加到集合或实体的服务) 与独立的微服务相匹配。
- 每个微服务应处理一个完整的业务功能。

领域模型又可以分为失血、贫血和充血三种。

- 失血模型: 基于数据库的领域设计方式就是典型的失血模型, 只关注数据的增删改查。
- 贫血模型: 在 `Domain Object` 中包含了不依赖于持久化的领域逻辑, 而那些依赖持久化的领域逻辑被分离到 `Server` 层。
- 充血模型: 充血模型跟贫血模型差不多, 不同的是如何划分业务逻辑, 也就是说, 大部分业务应该放到 `Domain Object` 里面, 而 `Service` 应该是很“薄”的一层。

2.1 设计原则之分层架构

同一公司使用统一应用分层, 以减少开发维护学习成本。应用分层看起来很简单, 但每个程序员都有自己的一套方法, 哪怕是初学者, 所以想实施起来并非那么容易。



最早接触的分层架构应该是我们最熟悉的 MVC (Model-View-Controller) 架构, 其将应用分成了模型、视图和控制层, 可以说引导了绝大多数开发者。而现在的应用 (包括框架) 中非常多架构设计都使用此模式。之后又演化出了 MVP (Model-View-Presenter) 和 MVVM (Model-View-ViewModel)。这些可以说都是随着技术的不断发展, 为了应对不同场景所演化出来的模型。而微服务的每个架构都可以再细分成领域模型, 下面看一下经典的领域模型架构。

它包括了 Domain、Service 和 Repositories。核心实体 (Entity) 和值对象 (Value Object) 应该在 Domain 层, 定义的领域服务 (Domain Service) 在 Service 层, 而针对实体和值对象的存储和查询逻辑都应该在 Repositories 层。值得注意的是, 不要把 Entity 的属性和行为分离到 Domain 和 Service 两层中去实现, 即所谓的贫血模型, 事实证明这样的实现方式会造成很大的维护问题。基于这种设计, 工程的结构可以构造为:

```
- MicroService-Sample/src/
    domain
    gateways
    interface
    repositories
    services
```

当然, 在微服务的架构中, 每个微服务不必严格遵照这样的规定, 切忌死搬硬套, 最重要的是理解业务。在不同的业务场合, 架构的设计可以适当地调整, 毕竟适合的架构一定要具有灵活性。

分层的原则如下。

➤ 文件夹分层法

应用分层采用文件夹方式的优点是可大可小、简单易用、统一规范, 可以包括 5 个项目, 也可以包括 50 个项目, 以满足所有业务应用的多种不同场景。

➤ 调用规约

在开发过程中, 需要遵循分层架构的约束, 禁止跨层次的调用。

➤ 下层为上层服务

以用户为中心, 以目标为导向。上层 (业务逻辑层) 需要什么, 下层 (数据访问层) 就提供什么, 而不是下层 (数据访问层) 有什么, 就向上层 (业务逻辑层) 提供什么。

➤ 实体层规约

Entity 是数据表对象, 不是数据访问层对象; DTO 是网络传输对象, 不是表现层对象; BO 是内存计算逻辑对象, 不是业务逻辑层对象, 不是只能给业务逻辑层使用。如果仅限定在本层



访问，则导致单个应用内大量没有价值的对象转换。以用户为中心来设计实体类，可以减少无价值重复对象和无用转换。

➤ U 型访问

下行时表现层是 Input，业务逻辑层是 Process，数据访问层是 Output。上行时数据访问层是 Input，业务逻辑层是 Process，表现层是 Output。

2.2 设计原则之统一通信协议

接口调用如果是远程调用，那么就构成了简单的分布式。最简单的远程接口实现方式是 Web Service 或 REST。当然一个合理的分布式应用不仅仅是远程接口调用这么简单，还需要有负载均衡、缓存等功能。实现分布式最简单的技术是 REST 接口，因为 REST 接口可以使用现存的各种服务器，比如使用负载均衡服务器和缓存服务器来实现负载均衡和缓存功能。

关于通信协议，不同的公司有不同的选择，但是建议同一公司内部使用统一的通信协议，比较典型的有 GRPC 和 BRPC。

GRPC 是一个高性能、开源和通用的 RPC 框架，面向移动和 HTTP/2 设计。目前提供 C、Java 和 Go 语言版本，分别是 GRPC、GRPC-Java、GRPC-Go，其中 C 版本支持 C、C++、Node.js、Python、Ruby、Objective-C、PHP 和 C#。

GRPC 基于 HTTP/2 标准设计，带来诸如双向流、流控、头部压缩、单 TCP 连接上的多复用请求等特性。这些特性使得其在移动设备上表现更好，更省电和节省空间占用。

与 GRPC 类似，BRPC 源自百度，目前支撑百度内部大约 75 万个同时在线的实例。

其实基于以上的几种选择都能够完成高效的开发，团队内部使用统一的标准，这样更有利于模块化和统一标准。

服务间的通信通过轻量级的 Web 服务，使用同步的 REST API 进行通信。在实际的项目应用中，一般推荐在查询时使用同步机制，在增删改时使用异步的方式，结合消息队列来实现数据的操作，以保证最终的数据一致性。

REST API 应为创建、检索、更新和删除操作使用标准 HTTP 动词，而且应特别注意操作是否幂等。

POST 操作可用于创建资源。POST 操作的明显特征是它不是幂等的。举例而言，如果使用 POST 请求创建资源，而且启动该请求多次，那么每次调用后都会创建一个新的唯一资源。

GET 操作必须是幂等的且不会产生意外结果。具体来讲，带有查询参数的 GET 请求不应用于更改或更新信息（而应使用 POST、PUT 或 PATCH）。

PUT 操作可用于更新资源。PUT 操作通常包含要更新的资源的完整副本，使该操作具有幂



等性。

PATCH 操作允许对资源执行部分更新。它们不一定是幂等的，具体取决于如何指定增量并应用到资源上。例如，如果一个 PATCH 操作表明一个值应从 A 改为 B，那么它就是幂等的。如果它已启动多次而且值已是 B，则没有任何效果。对 PATCH 操作的支持仍不一致。例如，Java EE7 中的 JAX-RS 中没有 @PATCH 注释。

DELETE 操作用于删除资源。删除操作是幂等的，因为资源只能删除一次。但是，返回代码不同，因为第一次操作将成功（200），而后续调用不会找到资源（204）。

2.3 设计原则之单一职责

设计原则很重要的一点就是简单，单一职责也就是我们经常所说的专人干专事。

一个单元（一个类、函数或者微服务）应该有且只有一个职责。无论如何，一个微服务不应该包含多于一个的职责。职责单一的后果之一就是责任单位（微服务、类、接口、函数）的数量剧增。据说 Amazon、Netflix 这些采用微服务架构的网站一个小功能就会调用几十上百个微服务。但是相较于每个函数都是多个业务逻辑或职责功能的混合体的情况，维护成本还是低很多的。

SRP（单一职责原则）中的“单一职责”是个比较模糊的概念。对于函数，它可能指单一的功能，不涉及复杂逻辑；但对于类或者接口，它可能是指对单一对象的操作，也可能是指对该对象单一属性的操作。总而言之，单一职责原则就是为了让代码逻辑更加清晰，可维护性更好，定位问题更快的一种设计原则。

单一职责的优点如下：

- 类的复杂性降低，实现什么职责都有清晰明确的定义。
- 可读性提高，复杂性降低。
- 可维护性提高，可读性提高。

变更引起的风险降低，变更是必不可少的，如果接口的单一职责做得好，一个接口修改只对相应的实现类有影响，对其他的接口无影响，这对系统的扩展性、维护性都有非常大的帮助。

记得在三字经里边有这样一段：

教之道，贵以专

说的就是无论学习还是构建团队，最重要的是专才，而不是全才。就好比一个足球队，如果都是前锋或者都是后卫，那么这样的球队一定不会出好成绩，反而是将各个位置上的人进行统一协调，根据分工不同，共同协作，形成 $1+1>2$ 的效果，那么这样的团队就非常容易



出好成绩。

有很多公司为了赶进度，经常会招聘一些所谓的全能型人才，但是这种人往往专业程度不够高，当遇到某些棘手的问题时，往往不能够非常快速地解决问题。从而导致最终交付的软件质量较差。

实施单一职责的目的如下：

- 以类来隔离需求功能点，这样当一个点的需求发生变化时，不会影响别的类的逻辑，这个和设计模式中的开闭原则类似，对于扩展持开放态度，对于修改持关闭态度。
- 是一个原子模块级的粒度，至于原子的粒度到底是什么样的，应该因业务而异，设计的过程中同时考虑业务的扩展，所以这就要求在设计的过程中，必须有业务专家共同参与，共同规避风险。
- 粒度小，灵活，复用性强，方便更高级别的抽象。

每个微服务单独运行在独立的进程中，能够实现松耦合，并且独立部署。

2.4 设计原则之服务拆分

拆分粒度不应该过分追求细粒度，要考虑适中，不能过大或过小。按照单一职责原则和康威定律，在业务域、团队和技术上平衡粒度。拆分后的代码应该是易控制、易维护的，业务职责也是明确单一的。

AKF 扩展立方体是一个叫 AKF 公司的技术专家抽象总结的应用扩展的三个维度。理论上按照这三个扩展模式，可以将一个单体系统进行无限扩展。AKF 扩展立方如图 2-1 所示。

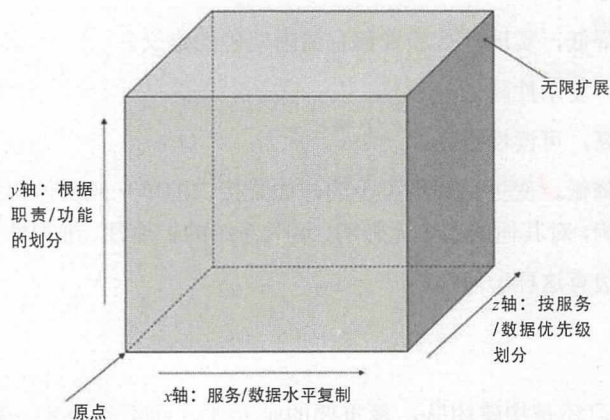


图 2-1 AKF 扩展立方体

- x 轴：水平复制，即在负载均衡服务器后增加多个 Web 服务器。



- y 轴：功能分解，将不同职能的模块分成不同的服务。从 y 轴方向扩展，才能将巨型应用分解为一组不同的服务，例如，订单管理中心、商品信息管理中心、库存管理中心等。
- z 轴：对数据库的扩展，即分库分表（分库是将关系紧密的表放在一台数据库服务器上，分表是因为一张表的数据太多，需要将一张表的数据通过 Hash 放在不同的数据库服务器上）。

三个维度的扩展对比如表 2-1 所示。

表 2-1 维度扩展对比

维 度	优 点	缺 点	场 景
x 轴扩展	成本最低，实施简单	受指令集多少和数据集大小的约束。当单个产品或应用过大时，服务响应变慢，无法通过 x 轴的水平扩展提高速度	发展初期，业务复杂度低，需要增加系统容量
y 轴扩展	可以解决指令集和数据集的约束，解决代码复杂度问题，可以实现隔离故障，可以提高响应时间，可以使团队聚焦更利于团队成长	成本相对较高	业务复杂，数据量大，代码耦合度高，团队规模大
z 轴扩展	能解决数据集的约束，降低故障风险，实现渐进交付，可以带来最大的扩展性	成本最昂贵，且不一定能解决指令集的问题	用户指数级快速增长

下面看一下 AKF 的拆分实践。

➤ 拆分应用

x 轴：从单体系统或服务，水平克隆出许多系统，通过负载均衡平均分配请求。

y 轴：面向服务分割，基于功能或者服务分割，例如，电商网站可以将登录、搜索、下单等服务进行 y 轴的拆分，每一组服务再进行 x 轴的扩展。

z 轴：面向查找分割，基于用户、请求或者数据分割，例如，可以将不同产品的 SKU 分到不同的搜索服务，可以将用户哈希到不同的服务等。

➤ 拆分数据库

x 轴：从单库水平克隆为多个库上读，一个库写，通过数据库的自我复制实现，要允许一定的读写时延。

y 轴：根据不同的信息类型分割为不同的数据库，即分库，例如，产品库、用户库等。

z 轴：按照一定算法进行分片，例如，将搜索按照 MapReduce 的原理进行分片，把 SKU 的数据按照不同的哈希值进行分片存储，每个分片再进行 x 轴冗余。



要做好微服务的分层：梳理和抽取核心应用、公共应用，作为独立的服务下沉到核心和公共能力层，逐渐形成稳定的服务中心，使前端应用能更快速地响应多变的市场需求。

对于服务的拆分，要使用迭代演进的方式，不能一次性完成所有的服务的拆分，需要确保团队可接受，粒度适中，同时需要优先考虑 API 的版本兼容性。不能够单纯以代码量来对服务拆分的成果进行评估。

2.5 设计原则之前后端分离

在传统的 Web 应用开发中，大多数的程序员会将浏览器作为前后端的分界线。将浏览器中为用户进行页面展示的部分称之为前端，而将运行在服务器，为前端提供业务逻辑和数据准备的所有代码统称为后端。

由于前后端分离这个概念相对来说刚出现不久，很多人都是只闻其声，不见其形，所以可能会对它产生一些误解，误以为前后端分离只是一种 Web 应用开发模式，只要在 Web 应用的开发期进行了前后端开发工作的分工就是前后端分离。

其实前后端分离并不只是开发模式，而是 Web 应用的一种架构模式。在开发阶段，前后端工程师约定好数据交互接口，实现并行开发和测试；在运行阶段前后端分离模式需要对 Web 应用进行分离部署，前后端之前使用 HTTP 或者其他协议进行交互请求。

前后端分离原则，简单来讲就是前端和后端的代码分离，也就是技术上做分离。推荐的模式是直接采用物理分离的方式部署，进一步进行更彻底的分离。不要继续使用以前的服务端模板技术，比如 JSP，把 Java JS HTML CSS 都堆到一个页面里，稍复杂的页面就无法维护。

这种分离模式的方式有几个好处：

- 前后端技术分离，可以由各自的专家来对各自的领域进行优化，这样前端的用户体验优化效果会更好。
- 分离模式下，前后端交互界面更加清晰，就剩下了接口和模型，后端的接口简洁明了，更容易维护。
- 前端多渠道集成场景更容易实现，后端服务无须变更，采用统一的数据和模型，可以支撑前端的 Web UI/移动 App 等访问。

前后端分离意味着前后端之间使用 JSON 来交流，两个开发团队之间使用 API 作为契约进行交互。从此，后台选用的技术栈不影响前台。

前后端分离并非仅仅只是前后端开发的分工，而是在开发期进行代码存放分离、前后端开发职责分离，前后端能够独立进行开发测试；在运行期进行应用部署分离，前后端之间通过 HTTP 请求进行通信。前后端分离的开发模式与传统模式相比，能为我们提升开发效率、增强

代码可维护性，让我们有规划地打造一个前后端并重的精益开发团队，更好地应对越来越复杂多变的 Web 应用开发需求。

前后端分离的核心：后台提供数据，前端负责显示。

2.6 设计原则之版本控制

在团队的内部，尤其是 API 设计优先的微服务架构中，接口的版本管理显得尤其重要。

微服务的一个主要优势是，允许服务独立地演变。考虑到微服务会调用其他服务，这种独立性需要引起高度注意，不能在 API 中制造破坏性更改。

接纳更改的最简单方法是绝不破坏 API。如果遵循稳健性原则，而且两端都保守地发送数据，慷慨地接收数据，那么可能很长一段时间都不需要执行破坏性更改。当最终发生破坏性更改时，可以选择构建一个完全不同的服务并不断退役原始服务，原因可能是领域模型已进化，而且一种更好的抽象更有意义。

如果对现有服务执行破坏性的 API 更改，应决定如何管理这些更改：

- 该服务是否会处理 API 的所有版本？
- 是否会维护服务的独立版本，以支持 API 的每个版本？
- 服务是否仅支持 API 的最新版本，依靠其他适应层来与旧 API 来回转换？

在确定了困难部分后，如何在 API 中反映该版本是更容易解决的问题。通常可通过 3 种方式处理 REST 资源的版本控制。

➤ 将版本放入 URI 中

将版本添加到 URI 中，这是指定版本的最简单方法。它的优点是非常容易理解，容易在微服务应用构建服务时实现，与 API 浏览工具和命令行工具兼容。如果将版本放在 URI 中，版本应该会应用于整个应用程序，所以使用 `/api/v1/accounts` 而不是 `/api/accounts/v1`。

➤ 使用自定义请求标头

可以添加自定义请求标头来表明 API 版本。在将流量路由到特定后端实例时，路由器和 other 基础架构可能会考虑使用自定义标头。但是，此机制不容易使用，原因与 `Accept` 标头不容易使用相同。此外，它是一个仅适用于应用程序的自定义标头，这意味着使用者需要学习如何使用它。

➤ 将版本放在 HTTP `Accept` 标头中，并依靠内容协商

`Accept` 标头是一个定义版本的明显位置，但它是最难测试的地方之一。URI 很容易指定和替换，但指定 HTTP 标头需要更详细的 API 和命令行调用。

2.7 设计原则之围绕业务构建

正所谓“不围绕业务构建的架构就是要流氓”。

微服务应当聚焦于某一特定的业务功能，并且确保完成它。其实这给需求管理也带来了挑战，需求需要切分将更加精细，以满足系统业务的不断变化。在传统的方式中，一般是产品人员进行需求调研，然后经过整理后提交给开发团队，这种方式在微服务的环境下需要重新定义，即产品核实需求后，需要在提交给开发团队之前进行评审，评审需要开发团队的人员参与，确认无误后再提交给开发团队。

从技术上说，微服务不应该局限于某个技术栈或者后端存储，可以非常灵活，以便于解决业务问题。这一点在非微服务的系统设计时，可能导致我们做一些妥协。而微服务可以让你用你认为最合适的方式解决问题。这和上面的统一框架并不冲突，统一是指构建团队的过程中，尽量保持统一的架构，从而降低交互和沟通所带来的额外成本。

系统可以根据业务切分为不同的子系统，子系统又可以根据重要程度切分为核心和非核心子系统。切分的目的就是当出现问题时，保证核心业务模块正常运行，不影响业务的正常操作。同时解决各个模块子系统间的耦合、维护及拓展性。方便单独部署，确保当一个系统出现问题时，不会出现连锁反应而导致整个系统瘫痪。

各个系统间合理地使用消息队列，解决系统或模块间的异步通信，实现高可用、高性能的通信系统。

2.8 设计原则之并发流量控制

大流量一般的衡量指标就是系统的 TPS（每秒事务量）和 QPS（每秒请求量）。其实这个没有一个绝对的标准，一般都是根据机器的配置情况而定的，如果当前配置不能应对请求量，那么就可以视为大流量了。

一般的应对方案包括：

➤ 缓存

预先准备好数据，减少对数据库的请求。

➤ 降级

如果不是核心链路，那么就把这个服务降级，保证主干畅通。

➤ 限流

在一定时间内把请求限制在一定范围内，保证系统不被冲垮，同时尽可能提升系统的吞吐量。限流的方式有几种，最简单的就是使用计数器，在一段时间内，进行计数，与阈值进行比

较，到了时间临界点，将计数器清 0。

应对并发，很重要的一点就是区分 CPU 密集型和 I/O 密集型。

➤ CPU 密集型 (CPU-bound)

CPU 密集型也叫计算密集型，指的是系统的硬盘、内存性能相对 CPU 要好很多。此时，系统运作大部分的状况是 CPU Loading 100%，CPU 要读/写 I/O（硬盘/内存），I/O 在很短的时间就可以完成，而 CPU 还有许多运算要处理，CPU Loading 很高。

CPU bound 的程序一般而言 CPU 占用率相当高。这可能是因为任务本身不太需要访问 I/O 设备，也可能是因为程序是多线程实现，因此屏蔽掉了等待 I/O 的时间。

➤ I/O 密集型 (I/O bound)

I/O 密集型指的是系统的 CPU 性能相对硬盘、内存要好很多。此时，系统运作大部分的状况是 CPU 在等 I/O（硬盘/内存）的读/写操作，此时 CPU Loading 并不高。

I/O bound 的程序一般在达到性能极限时，CPU 占用率仍然较低。这可能是因为任务本身需要大量 I/O 操作，而 pipeline 做得不是很好，没有充分利用处理器能力。

在微服务架构下，如果涉及不同类型的业务，需要根据资源的使用情况选用合适的处理资源。

2.9 设计原则之 CAP

CAP 原则又称 CAP 定理，指的是在一个分布式系统中，Consistency（一致性）、Availability（可用性）和 Partition tolerance（分区容错性），三者不可兼得，如图 2-2 所示。

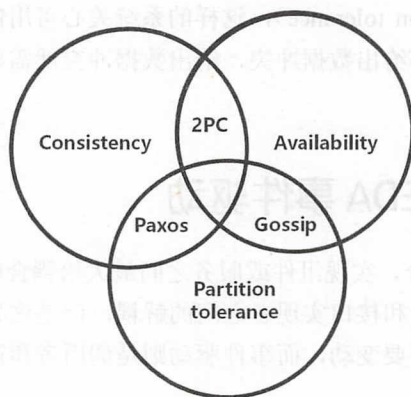


图 2-2 CAP 定理

下面对分布式系统中的三个特性进行了归纳。

- 一致性 (C): 在分布式系统中的所有数据备份在同一时刻是否有同样的值。
- 可用性 (A): 在集群中一部分节点故障后, 集群整体是否还能响应客户端的读写请求。
- 分区容忍性 (P): 以实际效果而言, 分区相当于对通信的时限要求。系统如果不能在时限内达成数据一致性, 就意味着发生了分区的情况, 必须就当前操作在 C 和 A 之间做出选择。

掌握 CAP 定理, 尤其是能够正确理解 C、A、P 的含义, 对于系统架构来说非常重要。因为对于分布式系统来说, 网络故障在所难免, 如何在出现网络故障时, 维持系统按照正常的行为逻辑运行就显得尤为重要。可以结合实际的业务场景和具体需求来进行权衡。

例如, 对于大多数互联网应用来说, 因为机器数量庞大, 部署节点分散, 网络故障是常态, 可用性是必须要保证的, 所以只有舍弃一致性来保证服务的 AP。而对于银行等需要确保一致性的场景, 通常会权衡 CA 和 CP 模型, CA 模型网络故障时完全不可用, CP 模型具备部分可用性。所以, 在设计微服务的时候一定要选择合适的模型。

CA (consistency + availability), 这样的系统关注一致性和可用性, 它需要非常严格的全体一致的协议, 比如“两阶段提交”(2PC)。CA 系统不能容忍网络错误或节点错误, 一旦出现这样的问题, 整个系统就会拒绝写请求, 因为它并不知道对面的那个结点是否挂掉了, 还是只是网络问题。唯一安全的做法就是把自己变成只读的。

CP (consistency + partition tolerance), 这样的系统关注一致性和分区容忍性。它关注的是系统里大多数人的 consistency 协议, 比如 Paxos 算法 (Quorum 类的算法)。这样的系统只需要保证大多数节点数据一致, 而少数的节点会在没有同步到最新版本的数据时变成不可用的状态, 这样能够提供一部分的可用性。

AP (availability + partition tolerance), 这样的系统关心可用性和分区容忍性。因此, 这样的系统不能达成一致性, 需要给出数据冲突, 给出数据冲突就需要维护数据版本, Dynamo 就是这样的系统。

2.10 设计原则之 EDA 事件驱动

EDA 是一种以事件为媒介, 实现组件或服务之间最大松耦合的方式。传统面向接口编程以接口为媒介, 实现调用接口者和接口实现者之间的解耦, 但是这种解耦程度不是很高, 如果接口发生变化, 则双方代码都需要变动, 而事件驱动则是调用者和被调用者互相不知道对方, 两者只和中间消息队列耦合。

在事件驱动的架构中, 跨服务完成业务逻辑的一个关键点是每个服务自动更新数据库和发布事件, 也就是要以原子粒度更新数据库和发布事件。

保证数据更新与事件发布原子化的方法有以下几种。

➤ 使用本地事务发布事件

一个实现原子化的方法是使用本地事务来更新业务实体和事件列表，由一个独立进程来发布事件。使用单独的事件表记录事件状态，然后使用单独的进程来监控事件的变化情况，确保事件实时发布。这种方式的缺点是数据更新和事件之间的对应关系是由开发者实现的，极有可能出错。

➤ 挖掘数据库事务日志

由线程或者进程通过挖掘数据库事务或提交日志来发布事件。应用更新数据库，数据库的事务日志会记录这些变更。事务日志挖掘线程或进程读取这些日志，并把事件发布到消息队列。这种方式的优点是不需要开发人员参与，缺点是与数据库紧耦合，需要根据数据库的变化而变化，另外根据数据库日志不一定能推断出所有的业务场景。

➤ 使用事件源

事件源采用一种截然不同的、以事件为中心的方法来保存业务实体——不同于存储实体的当前状态，应用存储的是状态改变的事件序列。每当业务实体的状态改变，新事件就被附加到事件列表，并且应用可以通过事件回放来重构实体的当前状态。事件长期保存在事件仓库(Event Store)，使用 API 添加和检索实体的事件。通过 API 让服务订阅事件，将所有事件传达到所有感兴趣的订阅者。所以，事件仓库可以认为是数据库与消息代理的综合体。缺点是想要构建一个高效的事件仓库并不容易。

2.11 设计原则之 CQRS

CQRS 是命令查询责任分离。

CQRS 架构由于本身只是一个读写分离的思想，实现方式多种多样。比如数据存储不分离，仅仅只是代码层面读写分离，也是 CQRS 的体现；数据存储的读写分离，C 端负责数据存储，Q 端负责数据查询，Q 端的数据通过 C 端产生的 Event 来同步，这种也是 CQRS 架构的一种实现。

传统架构的数据一般是强一致性的，我们通常会使用数据库事务保证一次操作的所有数据修改都在一个数据库事务里，从而保证了数据的强一致性。在分布式的场景中，我们也同样希望数据的强一致性，就是使用分布式事务。众所周知，分布式事务的难度、成本是非常高的，而且采用分布式事务的系统的吞吐量都会比较低，系统的可用性也会比较低。所以，很多时候，我们也会放弃数据的强一致性，而采用最终一致性；从 CAP 定理的角度来说，就是放弃一致性，选择可用性。

CQRS 架构则完全秉持最终一致性的理念。这种架构基于一个很重要的假设，就是用户看到的数据总是旧的。对于一个多用户操作的系统，这种现象很普遍。比如秒杀的场景，下单前，也许界面上显示的商品数量是有的，但是当你下单时，系统提示商品卖完了。其实我们只要仔细想想，也确实如此。因为我们在界面上看到的数据是从数据库取出来的，一旦显示到界面上就不会变了。但是很可能其他人已经修改了数据库中的数据。这种现象在大部分系统中，尤其是高并发的 Web 系统很常见。所以，基于这样的假设，我们知道，即便系统做到了数据的强一致性，用户还是很可能会看到旧的数据。这就给我们设计架构提供了一个新的思路。我们能否这样做：只需要确保系统的一切添加、删除和修改操作所基于的数据是最新的，而查询的数据不必是最新的。

微服务下 CQRS 的示例如图 2-3 所示。

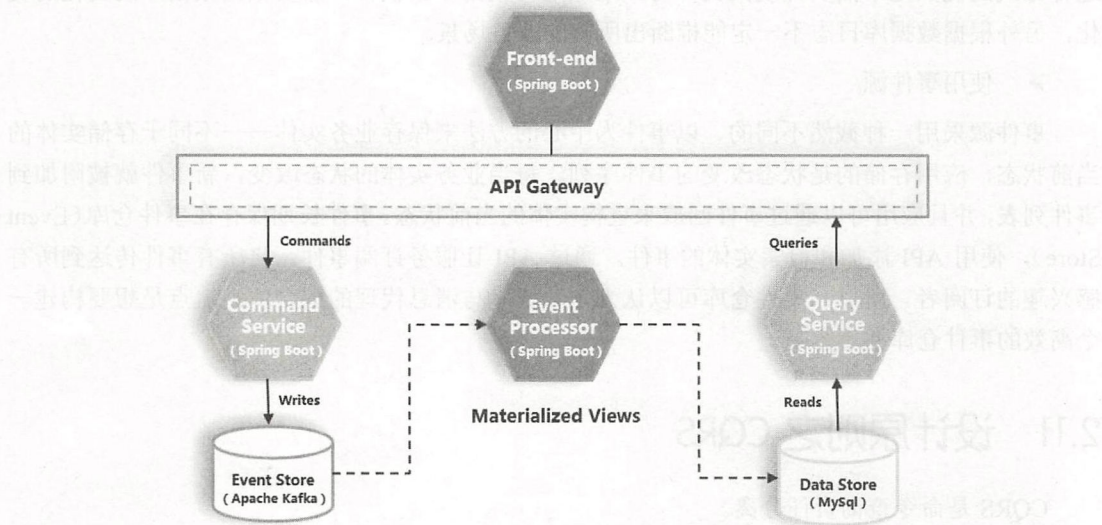


图 2-3 CQRS 示意图

将一个微服务分为命令端、查询端和事件处理器，这三个部分可以相互独立地部署。

➤ 命令端

请求采取命令的形式，可以驱动对微服务所拥有的领域数据的状态更改。

➤ 事件处理器

事件处理器可通过很多有用的方式对新的领域事件做出响应。一个领域事件可以生成多个事件，这些事件可以发送到其他微服务。

➤ 查询端

查询端将提供一个 REST API，允许 HTTP 客户端读取从已处理事件生成的实体化视图。

2.12 设计原则之基础设施自动化

微服务准备和构建的基础设施也是一个非常重要的需求。

基础设施应该包括提供给业务相关的应用所有基础保障的服务和设施，比如：

- DNS/CDN;
- 防火墙/Load Balancer;
- 应用服务器、数据库（物理机/虚拟机）;
- 日志、监控、报警服务。

一个服务应当被独立部署，并且包含所有的依赖、环境等物理资源。

服务足够小，功能单一，可以独立打包、部署、升级，不依赖其他服务，实现了局部自治，这就是我们应用架构的演进，从耦合到微服务，便于管理和服务的治理。

在传统的开发中，我们构建一个 WAR 包或 EAR 包，然后把它们部署在容器上。

而在一个规范的微服务中，每个微服务应该被构建成 fat JAR，其中内置了所有的依赖，然后作为一个单独的 Java 进程存在。

这里要单独说明一下，很多团队已经习惯了使用 WAR 包的方式发布，理所当然地认为发布就必须使用 WAR 包的方式，其实这完全取决于团队的自动化程度，当自动化程度非常高的时候，使用 JAR 包的方式发布是没有问题的，但是前提是整个团队的规范必须统一，有统一的标准。

自动化的原则包括：

- 能够毫不费力且可靠地重建基础设施中的任何元素。
- 可任意处理系统。可以轻松创建、销毁、替换、更改和移动资源。
- 一致的系統。假设两个基础设施元素提供相似的服务，比如同一个集群中有两个应用程序服务器，这些服务器应该几乎完全相同。它们的系统软件和配置应该是一样的，除了极少的用于区分彼此的配置（比如 IP 地址）。
- 可重复的过程。基于可再生原则，对基础设施执行的任何行为都是可以重复的。
- 变化的设计。确保系统能够安全地改变，迅速地做出变化。

2.13 设计原则之数据一致性

数据一致性分为以下几种情况。

➤ 强一致性

当更新操作完成之后，任何多个后续进程或者线程的访问都会返回最新的更新过的值。这种是对用户最友好的，就是用户上一次写什么，下一次就保证能读到什么。根据 CAP 理论，这种实现需要牺牲可用性。

➤ 弱一致性

系统并不保证后续进程或者线程的访问都会返回最新的更新过的值。系统在数据写入成功之后，不承诺立即可以读到最新写入的值，也不会具体地承诺多久之后可以读到。

➤ 最终一致性

弱一致性的特定形式。系统保证在没有后续更新的前提下，系统最终返回上一次更新操作的值。在没有故障发生的前提下，不一致窗口的时间主要受通信延迟、系统负载和复制副本的个数影响。DNS 是一个典型的最终一致性系统。

在工程实践上，为了保障系统的可用性，系统大多将强一致性需求转换成最终一致性的需求，并通过系统执行幂等性来保证数据的最终一致性。

微服务架构下，完整交易跨越多个系统运行，事务一致性是一个极具挑战的话题。依据 CAP 理论，必须在可用性（Availability）和一致性（Consistency）之间做出选择。我们认为在微服务架构下应选择可用性，然后保证数据的最终一致性。在实践中有三种模式：可靠事件模式、业务补偿模式和 TCC 模式。

- 可靠事件模式：可靠事件模式属于事件驱动架构，当某件重要的事情发生时，例如，更新一个业务实体，微服务会向消息代理发布一个事件，消息代理向订阅事件的微服务推送事件，当订阅这些事件的微服务接受此事件时，就可以完成自己的业务，可能会引发更多的事件发布。
- 业务补偿模式：补偿模式使用一个额外的协调服务来协调各个需要保证一致性的工作服务，协调服务按顺序调用各个工作服务，如果某个工作服务调用失败，则撤销之前所有已经完成的工作服务。要求需要保证一致性的工作服务提供补偿操作。
- TCC 模式：一个完整的 TCC 业务由一个主业务服务和若干个从业务服务组成，主业务服务发起并完成整个业务活动，TCC 模式要求从服务提供三个接口——Try（负责资源检查）、Confirm（真正执行业务）和 Cancel（释放 Try 阶段预留的资源）。

2.14 设计原则之设计模式

微服务的设计模式主要有以下几种：链式设计模式、聚合器设计模式、数据共享设计模式和异步消息控制模式。

➤ 链式设计模式

链式设计模式是常见的一种设计模式，用于微服务之间的调用，应用请求通过网关到达第一个微服务，微服务经过基础业务处理，发现不能满足要求，继续调用第二个服务，然后将多个服务的结果统一返回到请求中，如图 2-4 所示。

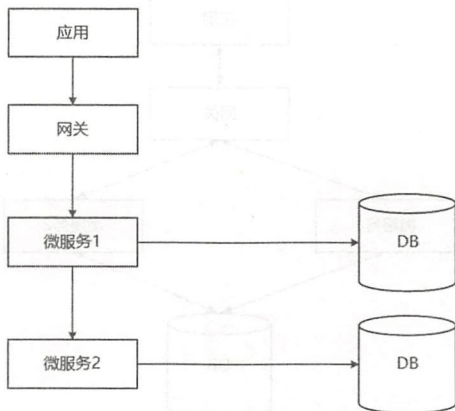


图 2-4 链式设计模式

➤ 聚合器设计模式

聚合器设计模式是将请求统一由网关路由到聚合器，聚合器向下路由到指定的微服务中获取结果，并且完成聚合，如图 2-5 所示。首页展现、分类搜索和个人中心等通常都使用这种设计。

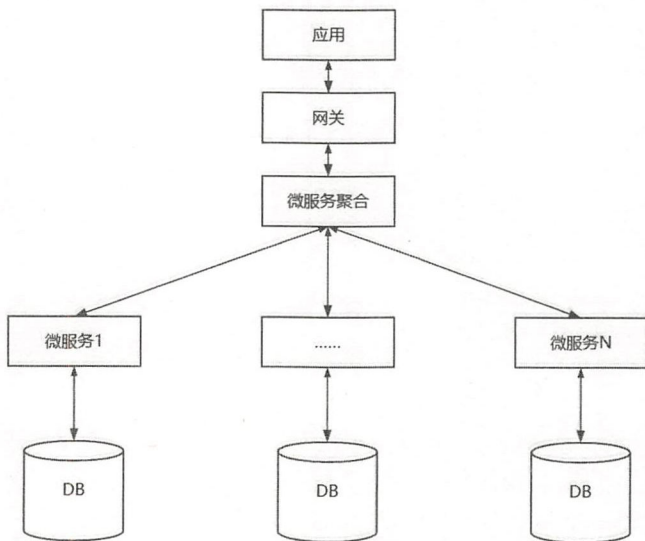


图 2-5 聚合器设计模式

➤ 数据共享模式

数据共享模式也是微服务设计模式的一种。应用通过网关调用多个微服务，微服务之间的数据共享通过同一个数据库，这样能够有效地减少请求次数，并且对于某些数据量小的情况非常适合，如图 2-6 所示。

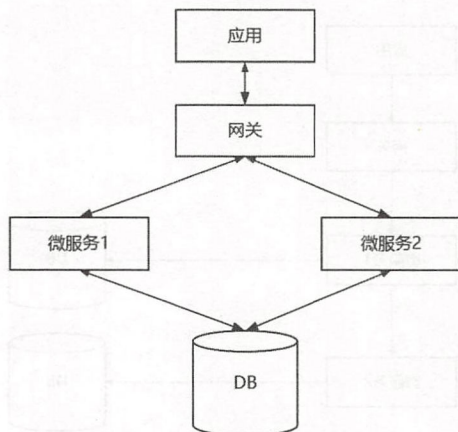


图 2-6 数据共享设计模式

➤ 异步消息设计模式

异步消息设计模式乍看起来跟聚合器的设计方式很像，唯一的区别就是网关和微服务之间的通信是通过消息队列而不是通过聚合器的方式来进行的，采用的是异步交互的方式，如图 2-7 所示。

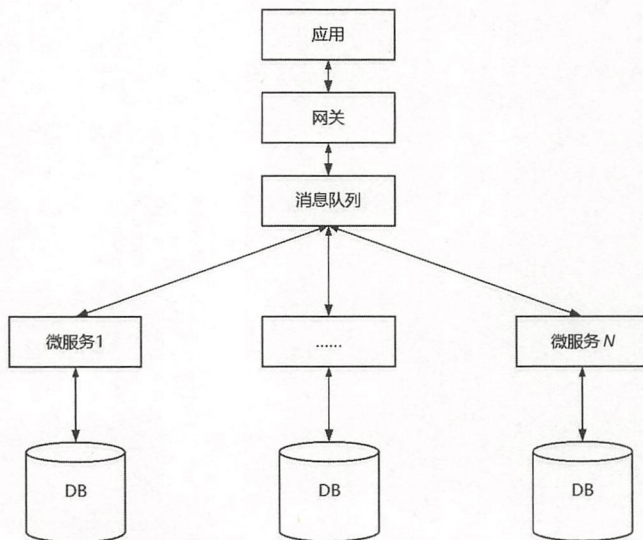


图 2-7 异步消息设计模式

2.15 设计原则之 DevOps

DevOps 一词来自 Development 和 Operations 的组合，突出重视软件开发人员和运维人员的沟通合作，通过自动化流程来使得软件构建、测试、发布更加快捷、频繁和可靠。它要求开发、测试、运维进行一体化的合作，进行更小、更频繁、更自动化的应用发布，以及围绕应用架构来构建基础设施的架构。这就要求应用充分内聚，也方便运维和管理。这个理念与微服务理念不谋而合。

DevOps 的出现是为了填补开发端和运维端之间的信息鸿沟，改善团队之间的协作关系。不过需要澄清的一点是，从开发到运维，中间还有测试环节。DevOps 其实包含了三个部分：开发、测试和运维。

换句话说，DevOps 希望做到的是软件产品交付过程中 IT 工具链的打通，使得各个团队减少时间损耗，更加高效地协同工作。

那么如何来评估 DevOps 的能力呢？可以通过能力环来对环境进行整体评估。DevOps 能力环如图 2-8 所示。

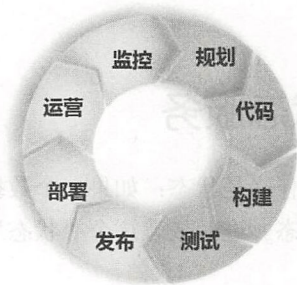


图 2-8 DevOps 能力环

无尽头的可能性：DevOps 涵盖了代码、部署目标的发布和反馈等环节，闭合成一个完整的 DevOps 能力闭环。

良好的闭环可以大大增加整体的产出。

执行 DevOps 的推荐工具如表 2-2 所示。

表 2-2 DevOps 推荐工具

服务注册与发现	部 署	监 控
ZooKeeper	Cloud Foundry	SonarCube
Doozer	Gradle	Logstash
etcd	Docker	New Relic
SmartStack	Docker Hub	Graphite

续表

服务注册与发现	部 署	监 控
Eureka	Docker Machine	Mesosphere / DCOS
NSQ	Kitematic	Winston
Serf	Docker Compose	Hystrix
Spotify	Docker Swarm	
DNS	AWS	
SkyDNS	Jenkins	
Consul	Continuum	
	Hudson	
	Artifactory	
	Terraform	
	Grunt	
	OpenShift	

不同的团队，情况都不相同——不同的技术栈，不同的团队文化，所以要选择适合自身团队的 DevOps 工具。

2.16 设计原则之无状态服务

对于无状态服务，首先说一下什么是状态：如果一个数据需要被多个服务共享，才能完成一笔交易，那么这个数据被称为状态。进而依赖这个“状态”数据的服务被称为有状态服务，反之称为无状态服务。

无状态服务原则并不是说在微服务架构里就不允许存在状态，其表达的真实意思是要把有状态的业务服务改变为无状态的计算类服务，那么状态数据也就相应地迁移到对应的“有状态数据服务”中。

场景说明：例如我们以前在本地内存中建立的数据缓存、Session 缓存，到现在的微服务架构中就应该把这些数据迁移到分布式缓存中存储，让业务服务变成一个无状态的计算节点。迁移后，就可以做到按需动态伸缩，微服务应用在运行时动态增删节点，就不再需要考虑缓存数据如何同步的问题。

无状态服务（Stateless Service）对单次请求的处理，不依赖其他请求，也就是说，处理一次请求所需的全部信息，要么都包含在这个请求里，要么可以从外部获取到（比如数据库），服务器本身不存储任何信息。Server 要设计为无状态的。

对服务器程序来说，究竟是有状态服务，还是无状态服务，其判断依旧是指两个来自相同

发起者的请求在服务器端是否具备上下文关系。如果是状态化请求，那么服务器端一般都要保存请求的相关信息，每个请求可以默认地使用以前的请求信息。而对于无状态请求，服务器端所能够处理的过程必须全部来自请求所携带的信息，以及其他服务器端自身所保存的，并且可以被所有请求所使用的公共信息。

无状态的服务器程序，最有名的就是 Web 服务器。每次 HTTP 请求和以前都没有关系，只是获取目标 URI。得到目标内容之后，这次连接就被“杀死”，没有任何痕迹。在后来的发展进程中，逐渐在无状态化的过程中，加入状态化的信息，比如 Cookie。服务端在响应客户端的请求时，会向客户端推送一个 Cookie，这个 Cookie 记录服务端上面的一些信息。客户端在后续的请求中，可以携带这个 Cookie，服务端可以根据这个 Cookie 判断请求的上下文关系。Cookie 的存在，是无状态化向状态化的一个过渡手段，通过外部扩展手段，Cookie 来维护上下文关系。

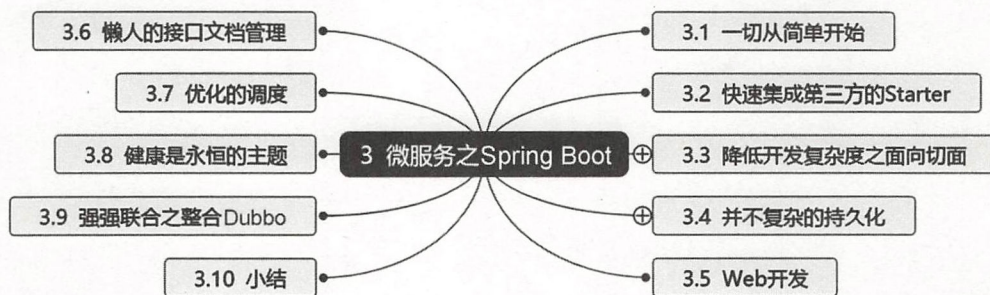
2.17 小结

以上整理了微服务常用的一些设计原则，通过这些原则的灵活运用，可以构建出一个高效的微服务系统。当然，微服务的构建是一个不断迭代、不断优化的过程，其最终目的是解决系统的性能瓶颈。也就是说，当发现系统出现瓶颈时，就需要考虑对系统基于一定的原则进行优化，提高系统的响应速度和可用性。



3 chapter

第 3 章 微服务之 Spring Boot



对于 Spring，相信大家都非常熟悉，从出现开始，一直是企业级开发的主流。但是随着软件的发展和应用开发的不断演化，它的一些缺点也逐渐暴露了出来。下面看一下 Spring 的发展历程并且认识一下 Spring Boot。

由来

在 Spring 1.x 的时候，所有的配置都通过 XML 实现，随着项目的扩大，需要频繁地在 Java 和 XML 之间切换。

在 Spring 2.x 的时候，已经开始逐步替换掉 XML 配置。在 Spring 3.x 的时候，已经开始提供 Java 的配置方式，在 Spring 4.x 的时候，已经全部推荐使用 Java 配置的方式。随着动态语言的流行，Java 的开发显得额外烦琐，主要体现在配置的复杂、开发效率低下，以及与第三方集成的繁杂等方面，这个时候 Spring Boot 应运而生。Spring 的发展历程如图 3-1 所示。

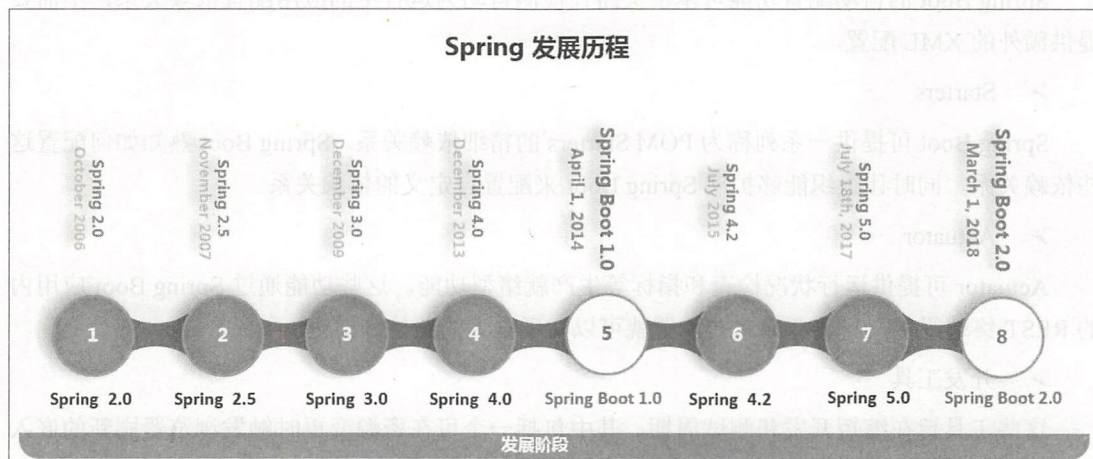


图 3-1 Spring 主要发展历程

Spring Boot 是由 Pivotal 团队提供的全新框架，其设计目的是用来简化 Spring 应用的初始搭建和开发过程。该框架使用了特定的方式来进行配置，从而使开发人员不再需要定义样板化的配置。简化了基于 Spring 的应用开发，通过少量的代码就能创建一个独立的、产品级别的 Spring 应用。Spring Boot 为 Spring 平台及第三方库提供开箱即用的设置，减少与第三方库集成的复杂度。

Spring Boot 的核心思想就是约定大于配置，多数 Spring Boot 应用只需要很少的 Spring 配置。采用 Spring Boot 可以大大简化开发模式，所有想集成的常用框架，它都有对应的组件支持。

特性

- (1) 方便地创建独立的 Spring 应用，为基于 Spring 的开发提供更快的入门体验。

(2) 内置 Tomcat，无须生成 WAR 文件。

(3) 简化 Maven 配置。

(4) 自动配置 Spring，更快、更方便地与第三方应用整合，比如消息队列、缓存等在企业级开发中常用的组件。

(5) 提供大型项目中的非功能特性，如指标、安全、健康检查及外部配置。

(6) 开箱即用，无须代码生成，也无须 XML 配置，同时能够通过修改默认值来满足待定的需求。

四大神器

➤ 自动配置

Spring Boot 的自动配置功能可基于类路径检测自动为运行中的应用配置依赖关系，不需要提供额外的 XML 配置。

➤ Starters

Spring Boot 可提供一系列称为 POM Starters 的精细依赖关系。Spring Boot 熟知如何配置这些依赖关系，同时让组织能够扩展 Spring Boot 来配置自定义的依赖关系。

➤ Actuator

Actuator 可提供运行状况检查和指标等生产就绪型功能。这些功能通过 Spring Boot 应用内的 REST 终端提供，只需要简单的配置就可以实现强大的监控和检查。

➤ 开发工具

这些工具旨在缩短开发和测试周期，其中包括一个可在资源变更时触发浏览器刷新的嵌入式 LiveReload 服务器。这些工具还提供了应用自动重启功能，只要类路径上的文件发生更改，该功能即可启动。重启技术使用两种类加载器，未更改的分类（例如，来自第三方 JAR 的类）被加载到基础类加载器，而开发中的分类则被加载到重启类加载器。当应用重启时，重启类加载器会被丢弃，同时创建一个新的类加载器。这种方法意味着应用重启的速度通常要比“冷启动”的速度快得多，因为基础类加载器已准备就绪且已填充完毕，从而快速实现应用的热部署，对于简单的修改这种场景能够非常有效地提高效率。

3.1 一切从简单开始

介绍

使用 Spring Boot 是快乐并且简单的，不需要烦琐的配置就能够完成一套非常强大的应用。

实现

使用 STS，可以去官方网站下载最新版。网站地址为 <https://spring.io/tools/sts/>。

Spring Tool Suite™是基于 Eclipse 开发的专门为 Spring 开发使用的工具包。

新建工程，选择 Spring Starter Project，如图 3-2 所示。

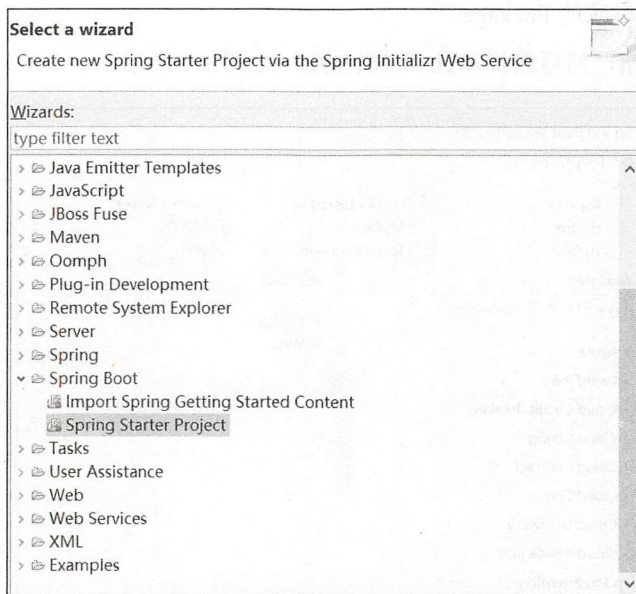


图 3-2 新建工程

单击“next”按钮，输入工程信息，如图 3-3 所示。

图 3-3 输入工程信息

- 输入工程名：对应的 Name。
- 打包方式：对应的 Packaging，可以选择 JAR 或者 WAR 的方式。
- 输入组织名：对应的 Group。
- 输入描述：对应的 Description。
- 输入包名：对应的 Package。

单击“next”按钮，然后选择 Web 和 MySQL，如图 3-4 所示。

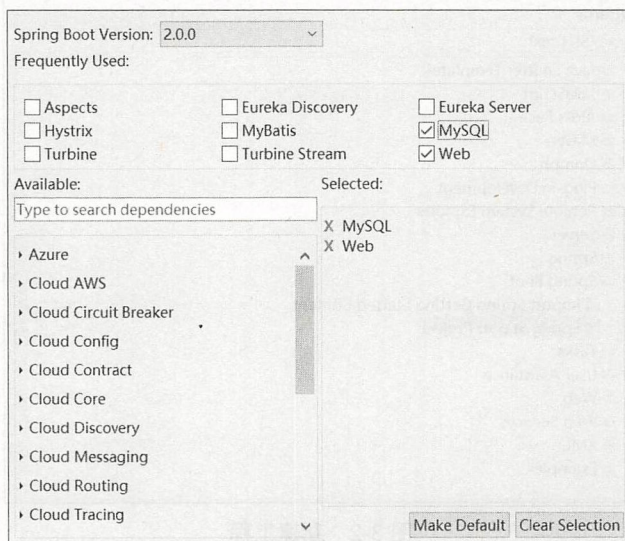


图 3-4 添加依赖

单击完成按钮后开始创建工程，创建完成后，Pom.xml 中会自动加入以下代码：

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.0.0.RELEASE</version>
  <relativePath /> <!-- lookup parent from repository -->
</parent>
```

进入 Chapter0301Application:

```
@RestController
@EnableAutoConfiguration
public class Chapter0301Application {
```

```

@RequestMapping("/")
String home() {
    return "欢迎使用 Spring Boot!";
}

public static void main(String[] args) {
    SpringApplication.run(Chapter0301Application.class, args);
}

```

`@RestController` 相当于 `@Controller` 和 `@RequestBody` 的结合, 是 Spring Boot 基于 Spring MVC 的基础上进行了改进, 将 `@Controller` 与 `@ResponseBody` 进行合并形成的一个新的注解。

`@EnableAutoConfiguration` 作用:

- 从 classpath 中搜索所有 META-INF/spring.factories 配置文件, 然后将其中的 org.springframework.boot.autoconfigure.EnableAutoConfiguration key 对应的配置项加载到 Spring 容器。
- 只有 spring.boot.enableautoconfiguration 为 true (默认为 true) 时, 才启用自动配置。
- `@EnableAutoConfiguration` 还可以进行排除, 排除方式有 2 种, 一是根据 class 来排除 (exclude), 二是根据 class name (excludeName) 来排除。

其内部实现的关键点有:

(1) ImportSelector, 该接口的方法的返回值都会被纳入 Spring 容器管理中。

(2) SpringFactoriesLoader, 该类可以从 classpath 中搜索所有 META-INF/spring.factories 配置文件, 并读取配置。

测试

配置完成后, 可以使用 main 函数的方式或者 Spring Boot App 的方式启动应用, 启动完成后可以看到如下输出:

```

:: Spring Boot ::                (v2.0.0.RELEASE)

2018-03-02 11:28:39.096 INFO 19144 --- [ restartedMain]
com.cloud.skyme.Chapter0301Application : Starting Chapter0301Application on
DESKTOP-E3I9LR5 with PID 19144
(C:\Users\zhangfeng\git\0301\chapter-03-01\target\classes started by
zhangfeng in C:\Users\zhangfeng\git\0301\chapter-03-01)

```



```
2018-03-02 11:28:39.097 INFO 19144 --- [ restartedMain]
com.cloud.skyme.Chapter0301Application : No active profile set, falling back
to default profiles: default
2018-03-02 11:28:39.241 INFO 19144 --- [ restartedMain]
ConfigServletWebServerApplicationContext : Refreshing
org.springframework.boot.web.servlet.context.AnnotationConfigServletWebServe
rApplicationContext@7785e155: startup date [Fri Mar 02 11:28:39 CST 2018]; root
of context hierarchy
2018-03-02 11:28:41.444 INFO 19144 --- [ restartedMain]
o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s):
8080 (http)
2018-03-02 11:28:41.487 INFO 19144 --- [ restartedMain]
o.apache.catalina.core.StandardService : Starting service [Tomcat]
2018-03-02 11:28:41.487 INFO 19144 --- [ restartedMain]
org.apache.catalina.core.StandardEngine : Starting Servlet Engine: Apache
Tomcat/8.5.28
2018-03-02 11:28:41.499 INFO 19144 --- [ost-startStop-1]
o.a.catalina.core.AprLifecycleListener : The APR based Apache Tomcat Native
library which allows optimal performance in production environments was not found
on the java.library.path:
[C:\java\Java\jdk1.8.0_131\bin;C:\Windows\Sun\Java\bin;C:\Windows\system32;C
:\Windows;C:\ProgramData\Oracle\Java\javapath;C:\Program Files
(x86)\Intel\iCLS Client\;C:\Program Files\Intel\iCLS
Client\;C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\S
ystem32\WindowsPowerShell\v1.0\;C:\Program Files (x86)\Intel\Intel(R)
Management Engine Components\DAL;C:\Program Files\Intel\Intel(R) Management
Engine Components\DAL;C:\Program Files (x86)\Intel\Intel(R) Management Engine
Components\IPT;C:\Program Files\Intel\Intel(R) Management Engine
Components\IPT;C:\java\apache-maven-3.5.0\bin;C:\java\Java\jdk1.8.0_131\bin;
C:\java\Java\jre8\bin;C:\Program Files\TortoiseGit\bin;C:\Program
Files\OpenVPN\bin;C:\Program Files\MATLAB\R2016a\runtime\win64;C:\Program
Files\MATLAB\R2016a\bin;C:\Program
Files\MATLAB\R2016a\polyspace\bin;C:\Python27\;C:\ProgramData\Anaconda2;C:\P
rogramData\Anaconda2\Scripts;C:\ProgramData\Anaconda2\Library\bin;C:\java\my
cat\bin;C:\Program Files\Git\cmd;C:\Program Files\TortoiseSVN\bin;C:\Program
Files\nodejs\;C:\Program Files
(x86)\Yarn\bin;$JAVA_HOME/bin;$JAVA_6_HOME/bin;$JAVA_7_HOME/bin;$JAVA_8_HOME
/bin;C:\Users\zhangfeng\AppData\Local\Microsoft\WindowsApps;;C:\Program
```

```
Files\Docke
Toolbox;C:\Users\zhangfeng\AppData\Roaming\npm;C:\Users\zhangfeng\AppData\Lo
cal\Yarn\bin;.]
2018-03-02 11:28:41.613 INFO 19144 --- [ost-startStop-1]
o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded
WebApplicationContext
2018-03-02 11:28:41.613 INFO 19144 --- [ost-startStop-1]
o.s.web.context.ContextLoader : Root WebApplicationContext:
initialization completed in 2376 ms
2018-03-02 11:28:41.781 INFO 19144 --- [ost-startStop-1]
o.s.b.w.servlet.ServletRegistrationBean : Servlet dispatcherServlet mapped to [/]
2018-03-02 11:28:41.787 INFO 19144 --- [ost-startStop-1]
o.s.b.w.servlet.FilterRegistrationBean : Mapping filter:
'characterEncodingFilter' to: [/]
2018-03-02 11:28:41.787 INFO 19144 --- [ost-startStop-1]
o.s.b.w.servlet.FilterRegistrationBean : Mapping filter:
'hiddenHttpMethodFilter' to: [/]
2018-03-02 11:28:41.787 INFO 19144 --- [ost-startStop-1]
o.s.b.w.servlet.FilterRegistrationBean : Mapping filter:
'httpPutFormContentFilter' to: [/]
2018-03-02 11:28:41.787 INFO 19144 --- [ost-startStop-1]
o.s.b.w.servlet.FilterRegistrationBean : Mapping filter:
'requestContextFilter' to: [/]
2018-03-02 11:28:42.202 INFO 19144 --- [ restartedMain]
s.w.s.m.m.a.RequestMappingHandlerAdapter : Looking for @ControllerAdvice:
org.springframework.boot.web.servlet.context.AnnotationConfigServletWebServe
rApplicationContext@7785e155: startup date [Fri Mar 02 11:28:39 CST 2018]; root
of context hierarchy
2018-03-02 11:28:42.298 INFO 19144 --- [ restartedMain]
s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/]}" onto public
java.lang.String com.cloud.skyme.controller.HelloController.index()
2018-03-02 11:28:42.308 INFO 19144 --- [ restartedMain]
s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/error]}" onto public
org.springframework.http.ResponseEntity<java.util.Map<java.lang.String,
java.lang.Object>>
org.springframework.boot.autoconfigure.web.servlet.error.BasicErrorControlle
r.error(javax.servlet.http.HttpServletRequest)
2018-03-02 11:28:42.309 INFO 19144 --- [ restartedMain]
```



```
s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped
"{[/error],produces=[text/html]}" onto public
org.springframework.web.servlet.ModelAndView
org.springframework.boot.autoconfigure.web.servlet.error.BasicErrorControlle
r.errorHtml(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpSer
vletResponse)
2018-03-02 11:28:42.365 INFO 19144 --- [ restartedMain]
o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/webjars/**] onto
handler of type [class
org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
2018-03-02 11:28:42.366 INFO 19144 --- [ restartedMain]
o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/**] onto handler
of type [class
org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
2018-03-02 11:28:42.411 INFO 19144 --- [ restartedMain]
o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/**/favicon.ico]
onto handler of type [class
org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
2018-03-02 11:28:42.586 INFO 19144 --- [ restartedMain]
o.s.b.d.a.OptionalLiveReloadServer : LiveReload server is running on port
35729
2018-03-02 11:28:42.666 INFO 19144 --- [ restartedMain]
o.s.j.e.a.AnnotationMBeanExporter : Registering beans for JMX exposure on
startup
2018-03-02 11:28:42.779 INFO 19144 --- [ restartedMain]
o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080
(http) with context path ''
2018-03-02 11:28:42.785 INFO 19144 --- [ restartedMain]
com.cloud.skyeme.Chapter0301Application : Started Chapter0301Application in
4.312 seconds (JVM running for 7.009)
```

输入 `http://localhost:8080`，可以看到输出相应的提示信息，如图 3-5 所示。

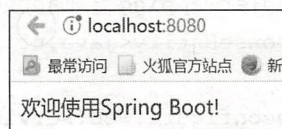


图 3-5 Spring Boot 启动页

给工程添加单元测试，在应用生成后，STS 会为应用自动生成基于 JUnit 的单元测试。添加的内容如下：

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes=Chapter0301Application.class,webEnvironment =
WebEnvironment.RANDOM_PORT)
public class Chapter0301ApplicationTests {

    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    public void testHome() {
        ResponseEntity<String> entity = this.restTemplate.getForEntity("/",
String.class);
        assertThat(entity.getStatusCode()).isEqualTo(HttpStatus.OK);
        assertThat(entity.getBody()).isEqualTo("欢迎使用 Spring Boot!");
    }
}
```

- `@RunWith(SpringRunner.class)` 让测试运行于 Spring 测试环境，此注释在 `org.springframework.test.annotation` 包中提供。
- `@SpringBootTest` 指定 Spring Boot 程序的测试引导入口。
- `TestRestTemplate` 是用于测试 REST 接口的模板类。

运行单元测试，测试上面构建的 Web 地址，可以看到输出的测试结果与期望的结果相同，如图 3-6 所示。

这样，一个 Web 应用从构建到单元测试就完成了，可见，构建一个 Spring Web MVC 的应用就是如此简单。

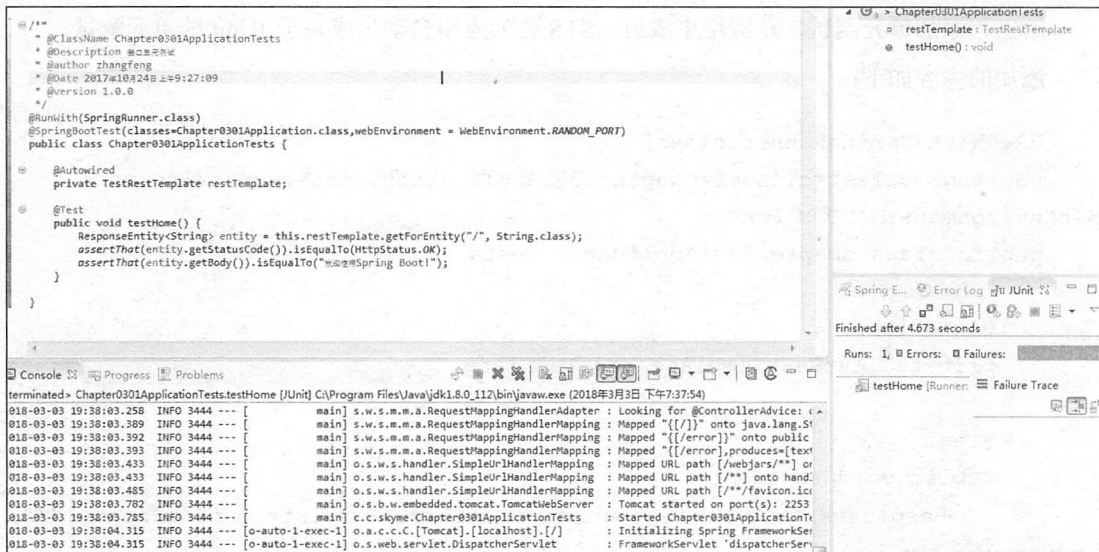


图 3-6 运行单元测试

再看看生成的依赖:

只需要在 `pom.xml` 中加入相应的启动器就可以,而这些都是已经由 STS 在构建工程的过程中为我们完成了。

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>Spring-boot-starter-web</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>Spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

总结

正如标题所说,一切从简单开始,时代的发展已经进入 AI 时代,所以开发也应该提速,能够智能化实现的部分就完全智能化。

3.2 快速集成第三方的 Starter

介绍

Spring Boot 的 Starter 主要用来简化依赖，对于企业级开发中与第三方的集成，可以通过一段简单的配置来完成，这样开发人员无须再对包依赖的问题头疼。Spring Boot 为我们提供了简化企业级开发的绝大多数场景的 Starter pom，只需要指定需要配置的 Starter，Spring Boot 会自动为我们提供配置好的 Bean。

原理

Spring Boot 常用的 Starter（启动器）如下。

- Spring-boot-starter-logging：使用 Spring Boot 默认的日志框架 Logback。
- Spring-boot-starter-log4j：添加 Log4j 的支持。
- Spring-boot-starter-web：支持 Web 应用开发，包含 Tomcat 和 Spring-mvc。
- Spring-boot-starter-tomcat：使用 Spring Boot 默认的 Tomcat 作为应用服务器。
- Spring-boot-starter-jetty：使用 Jetty 而不是默认的 Tomcat 作为应用服务器。
- Spring-boot-starter-test：包含常用的测试所需的依赖，如 JUnit、Hamcrest、Mockito 和 Spring-test 等。
- Spring-boot-starter-AOP：包含 Spring-AOP 和 AspectJ 来支持面向切面编程（AOP）。
- Spring-boot-starter-security：包含 Spring-security。
- Spring-boot-starter-jdbc：支持使用 JDBC 访问数据库。
- Spring-boot-starter-redis：支持使用 Redis。
- Spring-boot-starter-data-mongodb：包含 Spring-data-mongodb 来支持 MongoDB。
- Spring-boot-starter-data-jpa：包含 Spring-data-jpa、Spring-orm 和 Hibernate 来支持 JPA。
- Spring-boot-starter-amqp：通过 Spring-rabbit 支持 AMQP。
- Spring-boot-starter-actuator：添加适用于生产环境的功能，如性能指标和监测等功能。

当然，如果有必要，也可以定制自己的 Starter。关于完整的 Starter 可以到本书源码 chapter-03-02 中查看。

3.3 降低开发复杂度之面向切面

在软件业，AOP 为 Aspect Oriented Programming 的缩写，意为面向切面编程，通过预编译

方式和运行期动态代理实现程序功能的统一维护的一种技术。AOP 是 OOP 的延续，是软件开发中的一个热点，也是 Spring 框架中的一个重要内容，是函数式编程的一种衍生范型。利用 AOP 可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高了程序的可重用性，同时提高了开发的效率【此部分引自百度百科】

Spring AOP 的原理如图 3-7 所示。

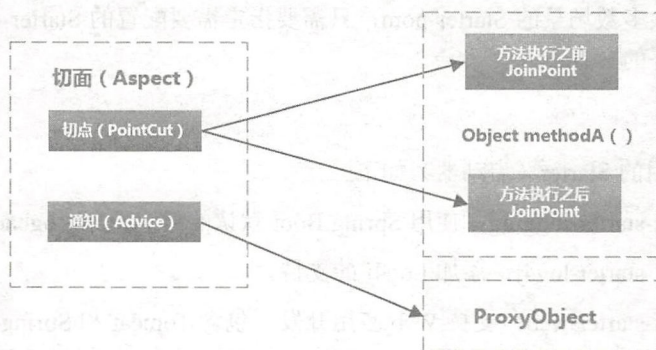


图 3-7 Spring AOP 的原理

AOP 把软件系统分为两个部分：核心关注点和横切关注点。业务处理的主要流程是核心关注点，与之关系不大的部分是横切关注点。横切关注点的一个特点是，它们经常发生在核心关注点的多处，而各处都基本相似，比如权限认证、日志、事务处理。AOP 的作用在于分离系统中的各种关注点，将核心关注点和横切关注点分离开来。

AOP 中的一些名词如下。

- 切面 (Aspect)：一个关注点的模块化，这个关注点可能会横切多个对象。事务管理是 J2EE 应用中一个关于横切关注点的很好的例子。在 Spring AOP 中，切面可以使用基于模式或者基于 @Aspect 注解的方式来实现。
- 连接点 (Joinpoint)：在程序执行过程中某个特定的点，比如某方法调用的时候或者处理异常的时候。在 Spring AOP 中，一个连接点总是表示一个方法的执行。
- 通知 (Advice)：在切面的某个特定的连接点上执行的动作，其中包括“around”、“before”和“after”等不同类型的通知（通知的类型将在后面部分进行讨论）。许多 AOP 框架（包括 Spring）都以拦截器作为通知模型，并维护一个以连接点为中心的拦截器链。
- 切入点 (Pointcut)：匹配连接点的断言。通知和一个切入点表达式关联，并在满足这个切入点的连接点上运行（例如，当执行某个特定名称的方法时）。切入点表达式如何和连接点匹配是 AOP 的核心：Spring 默认使用 AspectJ 切入点语法进行匹配。
- 引入 (Introduction)：用来给一个类型声明额外的方法或属性（也被称为连接类型声明

(inter-type declaration))。Spring 允许引入新的接口（以及一个对应的实现）到任何被代理的对象。例如，可以使用引入来使一个 Bean 实现 IsModified 接口，以便简化缓存机制。

- 目标对象(Target Object): 被一个或者多个切面所通知的对象, 也被称为被通知(advised)对象。既然 Spring AOP 是通过运行时代理实现的, 那么这个对象永远是一个被代理(proxyed)对象。
- AOP 代理(AOP Proxy): AOP 框架创建的对象, 用来实现切面契约(例如, 通知方法执行, 等等)。在 Spring 中, AOP 代理可以是 JDK 动态代理或者 CGLIB 代理。
- 织入(Weaving): 把切面连接到其他应用程序类型或者对象上, 并创建一个被通知的对象。这些可以在编译时(例如, 使用 AspectJ 编译器)、类加载时和运行时完成。Spring 和其他纯 Java AOP 框架一样, 在运行时完成织入。

3.3.1 前置通知

介绍

前置通知一般用在要执行的方法之前, 比如在业务中要对某些输入执行验证, 可以使用前置通知, 当验证不通过后可以不让业务继续向下执行或者阻断业务。

实现

(1) 在 STS 中新建 Spring Boot 工程 chapter-03-03, 选择包含 Aspects, 这样就会自动在工程中加入 AOP 依赖。主类上会自动加上 @SpringBootApplication。

@SpringBootApplication = (默认属性) @Configuration + @EnableAutoConfiguration + @ComponentScan。

@Configuration 的注解类标识这个类可以使用 Spring IoC 容器作为 Bean 定义的来源。@Bean 注解告诉 Spring, 一个带有 @Bean 的注解方法将返回一个对象, 该对象应该被注册为在 Spring 应用程序上下文中的 Bean。

@EnableAutoConfiguration: 能够自动配置 Spring 的上下文, 试图猜测和配置你想要的 Bean 类, 通常会自动根据类路径和 Bean 定义自动配置。

@ComponentScan: 会自动扫描指定包下的全部标有 @Component 的类, 并注册成 Bean, 当然包括 @Component 下的子注解 @Service、@Repository 和 @Controller。

加入 AOP 依赖后, 可以看到 Pom.xml 中已经增加如下内容:

```
<dependency>  org.springframework.boot:spring-boot-starter-aop:1.5.10
```



```

    <groupId>org.springframework.boot</groupId>
    <artifactId>Spring-boot-starter-aop</artifactId>
  </dependency>

```

(2) 日志框架的性能比较: Log4j < Logback < Log4j2, 所以我们在项目中直接整合 Log4j2。
添加 log4j2 依赖:

```

<!-- log related -->
<!-- exclude 掉 Spring-boot 的默认 log 配置 -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>Spring-boot-starter</artifactId>
  <exclusions>
    <exclusion>
      <groupId>org.springframework.boot</groupId>
      <artifactId>Spring-boot-starter-logging</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<!-- 引入 Log4j2 依赖 -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>Spring-boot-starter-log4j2</artifactId>
</dependency>
<!-- 加上这个才能辨认到 log4j2.yml 文件 -->
<dependency>
  <groupId>com.fasterxml.jackson.dataformat</groupId>
  <artifactId>jackson-dataformat-yaml</artifactId>
</dependency>
<!-- end of log related -->

```

(3) 在要增加 AOP 处理的类上添加如下内容:

```

@Component
@Aspect
public class WebControllerAop

```

@Aspect: 描述一个切面类, 定义切面类时需要打上这个注解。

@Configuration: Spring Boot 配置类。

(4) 定义切入点, 在 AOP 类的方法上加入 **Pointcut**。

```
@Pointcut("execution(* com.cloud.skyme.controller..*.*(..))")
public void executeService() {

}
```

@Pointcut: 声明一个切入点, 切入点决定了连接点关注的内容, 使得我们可以控制通知什么时候执行。**Spring AOP** 只支持 **Spring Bean** 的方法执行连接点。所以你可以把切入点看作 **Spring Bean** 上方法执行的匹配。一个切入点声明有两个部分: 一个包含名字和任意参数的签名, 还有一个切入点表达式, 该表达式决定了我们关注哪个方法的执行。

切入点表达式的格式: **execution**([可见性] 返回类型 [声明类型].方法名(参数) [异常])
[]中的为可选, 其他的还支持通配符的使用。

- *: 匹配所有字符;
- ..: 一般用于匹配多个包、多个参数;
- +: 表示类及其子类;
- 运算符有&&、||、!。

在上面的示例中匹配 **com.cloud.skyme.controller** 包及其子包下的所有类的所有方法。

(5) 在 AOP 中添加前置通知要执行的内容。

```
@Before("executeService()")
public void doBeforeAdvice(JoinPoint joinPoint)
```

@Before: 前置通知——在某连接点之前执行的通知, 但这个通知不能阻止连接点之前的执行流程 (除非它抛出一个异常)。

3.3.2 后置返回通知

后置返回通知一般是在业务处理完成, 需要给用户返回统一的结果时触发, 触发的方式和前置通知基本相同。

参考代码如下:

```
@AfterReturning(value = "execution(* com.cloud.skyme.controller..*.*(..))",
```



```
returning = "keys")
    public void doAfterReturningAdvice1(JoinPoint joinPoint, Object keys) {
        logger.info("第一个后置返回通知的返回值: "+keys);
    }
```

@AfterReturning: 后置通知——在某连接点正常完成后执行的通知，通常在一个匹配的方法返回时执行。

3.3.3 后置异常通知

后置异常通知一般在统一业务异常处理时使用，当业务收到某种异常时进行相应的处理。

后置异常通知的配置方式如下：

```
@AfterThrowing(value = "executeService()", throwing = "exception")
public void doAfterThrowingAdvice(JoinPoint joinPoint, Throwable exception) {
    //目标方法名
    logger.info(joinPoint.getSignature().getName());
    if(exception instanceof NullPointerException){
        logger.info("发生了空指针异常!!!!");
    }
}
```

@AfterThrowing: 异常通知——在方法抛出异常退出时执行的通知。

3.3.4 后置最终通知

后置最终通知的配置方式如下：

```
@After("executeService()")
public void doAfterAdvice(JoinPoint joinPoint){
    logger.info("后置最终通知执行了!!!!");
}
```

@After: 最终通知——当某连接点退出时执行的通知（不论是正常返回还是异常退出）。

3.3.5 环绕通知

后置最终通知的配置方式如下：

```
@Around("execution(* com.cloud.skyme.controller.*.testAround*(..))")
public Object doAroundAdvice(ProceedingJoinPoint proceedingJoinPoint){
    logger.info("环绕通知的目标方法名: "+proceedingJoinPoint.getSignature().getName());
    try {
        Object obj = proceedingJoinPoint.proceed();
        return obj;
    } catch (Throwable throwable) {
        throwable.printStackTrace();
    }
    return null;
}
```

@Around: 环绕通知——包围一个连接点的通知，如方法调用，这是最强大的一种通知类型。环绕通知可以在方法调用前后完成自定义的行为。它也会选择是否继续执行连接点，或直接返回它自己的返回值，或抛出异常来结束执行。

测试

运行 JUnit 测试，可以发现返回结果与我们期望的结果相同。

```
2018-03-02 13:21:00,385:INFO main (StartupInfoLogger.java:59) - Started
Chapter0303ApplicationTests in 13.122 seconds (JVM running for 16.359)
2018-03-02 13:21:01,327:INFO main (WebControllerAop.java:71) - 我是前置通知!!!
2018-03-02 13:21:01,330:INFO main (WebControllerAop.java:79) - testBeforeService
2018-03-02 13:21:01,330:INFO main (WebControllerAop.java:81) - com.cloud.
skyme.controller.AopTestController
2018-03-02 13:21:01,335:INFO main (HttpAspect.java:68) - url=/aop/
testBeforeService.do
2018-03-02 13:21:01,336:INFO main (HttpAspect.java:70) - method=GET
2018-03-02 13:21:01,337:INFO main (HttpAspect.java:72) - ip=127.0.0.1
2018-03-02 13:21:01,337:INFO main (HttpAspect.java:74) - class=com.cloud.
skyme.controller.AopTestController and method name = testBeforeService
2018-03-02 13:21:01,338:INFO main (HttpAspect.java:76) - 参数={}
2018-03-02 13:21:01,347:INFO main (HttpAspect.java:86) - url =
http://localhost/aop/testBeforeService.do end of execution
```



```
2018-03-02 13:21:01,348:INFO main (HttpAspect.java:95) - response=前置通知测试
```

```
2018-03-02 13:21:01,348:INFO main (WebControllerAop.java:126) - 后置最终通知执行了!!!!
```

```
2018-03-02 13:21:01,349:INFO main (WebControllerAop.java:95) - 第一个后置返回通知的返回值: 前四通知测试
```

```
2018-03-02 13:21:01,350:INFO main (WebControllerAop.java:100) - 第二个后置返回通知的返回值: 前四通知测试
```

```
2018-03-02 13:21:01,449:INFO Thread-5 (AbstractApplicationContext.java:989) - Closing org.springframework.web.context.support.GenericWebApplicationContext-@eda25e5: startup date [Fri Mar 02 13:20:47 CST 2018]; root of context hierarchy
```

3.3.6 AOP 总结

AOP 为我们提供了一种以切面方式编程的方法，在业务处理过程中，日志的记录、数据源的切换、事务的处理都可以用 AOP 的方式解决，而跟 Spring Boot 的整合，让我们开发 AOP 又进一步地简化，使面向对象能够更加方便地发挥所长。

3.4 并不复杂的持久化

数据的访问或者跟数据库的交互在任何应用中都是非常重要的一个环节。我们的应用中有可能使用关系型数据库，也有可能使用非关系型数据库。非关系型数据库也就是我们经常说的 NoSQL (Not Only SQL)。简言之，就是说在一个应用中，可以根据业务自身的特点和需要，选择适合的数据库，除了常用的关系型数据库，使用非关系型数据库也是不错的选择。

介绍

持久化，即把数据（如内存中的对象）保存到可永久保存的存储设备中（如磁盘）。持久化的主要应用是将内存中的对象存储到数据库中，或者存储到磁盘文件中。持久化是将程序数据在持久状态和瞬时状态间转换的机制。JDBC 和文件 I/O 操作就是一种典型的持久化机制。

Spring Boot 使持久化集成变成了一项非常简单的任务，因为它具有自动配置 Spring Data 以访问数据库的能力。只需在工程中将 Spring-boot-starter-data-jpa 包含进来，Boot 的自动配置引擎就能探测到工程需要数据访问功能，并且会在 Spring 应用上下文中创建必要的 Bean。

JPA 是 Java Persistence API 的简称，中文名是 Java 持久层 API，是 JDK 5.0 注解或 XML 描述对象一关系表的映射关系，并将运行期的实体对象持久化到数据库中。严格来说，它是一套持久层的规范，用作为持久化的统一标准接口。

Spring Data JPA 是 Spring 为 JPA 提供的一套实现,可以极大地简化 JPA 的写法,可以在几乎不用写实现的情况下,实现对数据的访问和操作。除 CRUD 外,还包括如分页、排序等一些常用的功能。

Mybatis 和 Hibernate 都对 JPA 有相应的实现,使用此规范的实现,可以极大地简化开发,这也是微服务所提倡的。

下面我们就根据上面提到的有关规范,来看一下 Spring Boot 是如何帮助我们简化开发的。

3.4.1 单数据源

介绍

在操作数据源的时候,我们一般会选择一个 ORM 框架,像 Mybatis、Hibernate 这种,这里我们选择 Mybatis 来进行处理。单数据源如图 3-8 所示。

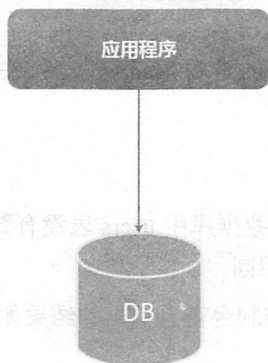


图 3-8 单数据源

实现

新建工程 chapter-03-04-01,添加数据源和 AOP 的依赖,然后在 pom.xml 中添加整合 Mybatis 的 Starter。

```
<dependency>
  <groupId>org.mybatis.spring.boot</groupId>
  <artifactId>mybatis-spring-boot-starter</artifactId>
  <version>1.3.1</version>
</dependency>
```

在 application.properties 中添加数据源配置,如果在正式的项目中,需要配置连接池,连接

池的选择包括 dbcp、c3p0 和 druid 等。

```
mybatis.type-aliases-package=com.cloud.skyme.entity

Spring.datasource.driverClassName = com.mysql.jdbc.Driver
Spring.datasource.url = jdbc:mysql://localhost:3306/users?useUnicode=
true&characterEncoding=utf-8
Spring.datasource.username = root
Spring.datasource.password = root
```

最后在启动文件上添加如下内容：

```
@SpringBootApplication
@MapperScan("com.cloud.skyme.mapper")
public class Chapter030401Application
```

@MapperScan 自动扫描所有 Mybatis 的 Mapper 文件，在这里扫描的是无 XML 形式的，XML 形式的扫描与其大致相同。

测试

运行 JUnit 测试 testInsert，确保数据库中 users 表没有数据，运行完成后，可以在数据库中看到结果，并且与期望的测试结果相同。

运行 JUnit 测试 testQuery，在控制台查看输出的结果如下：

```
2017-11-13 09:31:33.968 INFO 8316 --- [           main] com.cloud.skyme.
mapper.UserMapperTest    : users userName aa, pasword a123456sex MAN
2017-11-13 09:31:33.968 INFO 8316 --- [           main] com.cloud.skyme.
mapper.UserMapperTest    : users userName bb, pasword b123456sex WOMAN
2017-11-13 09:31:33.968 INFO 8316 --- [           main] com.cloud.skyme.
mapper.UserMapperTest    : users userName cc, pasword b123456sex WOMAN
```

同样，运行 JUnit 测试 testUpdate，从数据库中获取要更新的 id，然后运行测试。可以看到更新完成的结果与期望的结果相同。



3.4.2 多数据源

介绍

在一般的业务场景中，可能需要同时操作多个数据源。Spring Boot 也为多数据源的使用进行了优化。

多数据源如图 3-9 所示。

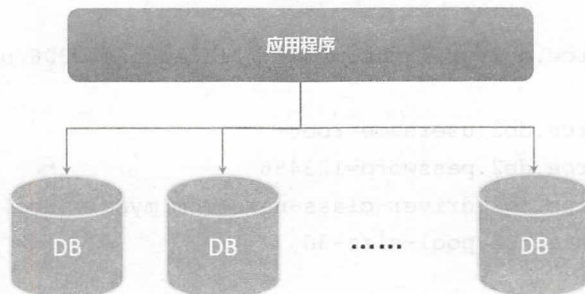


图 3-9 多数据源

实现

(1) 使用 STS 新建工程 chapter-03-04-02，然后在数据库中建立两个数据库，users 库中有用户信息，users1 库中信息与 users 信息相同，只是多了一个 flag 字段。

首先要将 Spring Boot 自带的 DataSourceAutoConfiguration 禁用，因为它会读取 application.properties 文件的 Spring.datasource.* 属性并自动配置单数据源。在 @SpringBootApplication 注解中添加 exclude 属性即可：

```

@SpringBootApplication(exclude = {
    DataSourceAutoConfiguration.class
})
public class Chapter030402Application {

    public static void main(String[] args) {
        SpringApplication.run(Chapter030402Application.class, args);
    }
}
  
```

(2) 在 application.properties 中添加数据源连接，生产环境中请使用连接池。




```
# user 源
spring.datasource.db1.url=jdbc:mysql://localhost:3306/users?characterEnc
oding=UTF-8
spring.datasource.db1.username=root
spring.datasource.db1.password=123456
spring.datasource.db1.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.db1.pool-size=30

# customer 源
spring.datasource.db2.url=jdbc:mysql://localhost:3306/users1?characterEn
coding=UTF-8
spring.datasource.db2.username=root
spring.datasource.db2.password=123456
spring.datasource.db2.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.db2.pool-size=30
```

(3) 由于我们禁用了自动数据源配置，因此下一步就需要手动将这些数据源创建出来。新建 `DataSourceConfig` 类并配置数据源信息。

```
@Configuration
public class DataSourceConfig {
    @Bean(name = "db1")
    @ConfigurationProperties(prefix = "Spring.datasource.db1") //
    application.properties 中对应属性的前缀
    public DataSource dataSource1() {
        return DataSourceBuilder.create().build();
    }
    @Bean(name = "ds2")
    @ConfigurationProperties(prefix = "Spring.datasource.db2") //
    application.properties 中对应属性的前缀
    public DataSource dataSource2() {
        return DataSourceBuilder.create().build();
    }
}
```

`@ConfigurationProperties` 读取配置文件中的指定值。

(4) 分别为两个 `Mapper` 指定各自的数据源，拿其中的一个数据源示例如下。



```

@Configuration
@MapperScan(basePackages = {"com.cloud.skyme.mapper.d1"},
sqlSessionFactoryRef = "sqlSessionFactory1")
public class MybatisDbAConfig {

    @Autowired
    @Qualifier("db1")
    private DataSource db1;

```

同理，另一个数据源的注入方式与上述这个相同。

(5) 配置完成后，将 TUserMapper 和 TUserMapper1 注入 UserController。

测试

启动应用，请求地址为 <http://localhost:8080/getAll>。

可以看到，数据从两个数据源中分别读取数据。

```

2017-11-13 09:42:57.744 INFO 21100 --- [nio-8080-exec-1] com.cloud.skyme.
web.UserController      : 数据源一 王五
2017-11-13 09:42:57.744 INFO 21100 --- [nio-8080-exec-1] com.cloud.skyme.
web.UserController      : 数据源二 张三

```

3.4.3 JOOQ

介绍

JOOQ 是基于 Java 访问的关系型数据库的工具包。JOOQ 既吸取了传统 ORM 操作数据的简单性和安全性，又保留了原生 SQL 的灵活性，它更像是 ORMS 和 JDBC 的中间层。

JOOQ 的优点：

- DSL (Domain Specific Language) 风格，代码简单和清晰。遇到不会写的 SQL 可以充分利用 IDEA 代码提示功能。
- 保留了传统 ORM 的优点，操作简单、类型安全等。不需要复杂的配置，并且可以利用 Java 8 Stream API 做更加复杂的数据转换。
- 支持主流的 RDMS 和更多的特性，如 self-joins, union、存储过程、复杂的子查询等。
- 丰富的 Fluent API 和完善的文档。
- runtime schema mapping 可以支持多个数据库 schema 访问。简单来说，使用一个连接



池可以访问多个 DB schema，使用比较多的就是 SaaS 应用的多租户场景。

实现

(1) 新建工程 chapter-03-04-03，并且加入 JOOQ 依赖。

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jooq</artifactId>
</dependency>
```

(2) 新建数据库，并且建立表 author。

(3) 到 C:\java\library 路径下准备 jooq-3.10.5.jar、jooq-codegen-3.10.5.jar、jooq-meta-3.10.5.jar、mysql-connector-java-5.1.45.jar 文件，并且编写文件 library.xml，文件内容如下：

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.9.2.xsd">
  <!-- Configure the database connection here -->
  <jdbc>
    <driver>com.mysql.jdbc.Driver</driver>
    <url>jdbc:mysql://localhost:3306/library?useUnicode=true&
characterEncoding=UTF-8</url>
    <user>root</user>
    <password>123456</password>
  </jdbc>

  <generator>
    <!-- The default code generator. You can override this one, to generate
your own code style.
Supported generators:
- org.jooq.util.JavaGenerator
- org.jooq.util.ScalaGenerator
Defaults to org.jooq.util.JavaGenerator -->
    <name>org.jooq.util.JavaGenerator</name>

  <database>
    <!-- The database type. The format here is:
org.util.[database].[database]Database -->
    <name>org.jooq.util.mysql.MySQLDatabase</name>
```



```

<!-- The database schema (or in the absence of schema support, in your
RDBMS this
    can be the owner, user, database name) to be generated -->
<inputSchema>library</inputSchema>

<!-- All elements that are generated from your schema
    (A Java regular expression. Use the pipe to separate several expressions)
    Watch out for case-sensitivity. Depending on your database, this
might be important! -->
<includes>.*</includes>

<!-- All elements that are excluded from your schema
    (A Java regular expression. Use the pipe to separate several expressions).
    Excludes match before includes, i.e. excludes have a higher priority -->
<excludes></excludes>
</database>

<target>
    <!-- The destination package of your generated classes (within the
destination directory) -->
    <packageName>com.cloud.skyme</packageName>

    <!-- The destination directory of your generated classes. Using Maven
directory layout here -->
    <directory>C:/workspace/MySQLTest/src/main/java</directory>
</target>
</generator>
</configuration>

```

运行生成命令：

```
java -classpath jooq-3.10.5.jar;jooq-meta-3.10.5.jar;jooq-codegen-3.10.5.jar;
mysql-connector-java-5.1.45.jar; org.jooq.util.GenerationTool library.xml
```

将生成的文件复制到工程中。

(4) 编写 service 类调用 JOOQ 的查询，代码如下：




```

try (Connection conn = DriverManager.getConnection(url, userName, password)) {
    DSLContext create = DSL.using(conn, SQLDialect.MYSQL);
    Result<Record> result = create.select().from(AUTHOR).fetch();

    for (Record r : result) {
        Integer id = r.getValue(AUTHOR.ID);
        String firstName = r.getValue(AUTHOR.FIRST_NAME);
        String lastName = r.getValue(AUTHOR.LAST_NAME);

        System.out.println("ID: " + id + " first name: " + firstName
+ " last name: " + lastName);
    }

    conn.close();
}

```

`org.jooq.impl.DSL` 是生成所有 jOOQ 对象的主要类。它作为一个静态的工厂去生成数据库表达式、列表表达式、条件表达式和其他查询部分。jOOQ 2.0 以后，为了使客户端代码更加趋近于 SQL，引进了静态工厂方法。在使用 DSL 时，只需要简单地从 DSL class 引入所有静态方法即可。

```

' Table<Record> table = create.select().from(AUTHOR).fetch(); 获取表记录对象
DSLContext create = DSL.using(conn, SQLDialect.MYSQL); 获取数据库连接

```

DSLContext 引用了 `org.jooq.Configuration`。Configuration 配置了 jOOQ 的行为，当 jOOQ 执行查询时，DSLContext 和 DSL 不同，DSLContext 允许创建已经配置和准备执行的 SQL 语句。

测试

启动应用，并且输入测试地址：<http://localhost:8080/jooq>。

可以看到如下输出：

```

Thank you for using jOOQ 3.10.5

ID: 1 first name: 3 last name: zhang
ID: 2 first name: 4 last name: li

```

证明调用成功。

3.4.4 事务处理

事务管理对于企业应用来说是至关重要的，即使出现异常情况，它也可以保证数据的一致性。Spring Framework 对事务管理提供了一致的抽象，包括：

- 为不同的事务 API 提供一致的编程模型，比如 JTA(Java Transaction API)、JDBC、Hibernate、JPA(Java Persistence API 和 JDO(Java Data Objects)。
- 支持声明式事务管理，特别是基于注解的声明式事务管理，简单易用。
- 提供比其他事务 API（如 JTA）更简单的程式化事务管理 API。
- 与 Spring 数据访问抽象的完美集成。

而我们所说的事务的关键属性，也就是我们经常说的 ACID 包括：

- 原子性（atomicity）——事务是一个原子操作，由一系列动作组成。事务的原子性确保动作要么全部完成，要么完全不起作用。
- 一致性（consistency）——一旦所有事务动作完成，事务就被提交。数据和资源就处于一种满足业务规则的一致性状态中。
- 隔离性（isolation）——可能有许多事务同时处理相同的数据，因此每个事务都应该与其他事务隔离开来，防止数据损坏。
- 持久性（durability）——一旦事务完成，无论发生什么系统错误，它的结果都不应该受到影响。通常情况下，事务的结果被写到持久化存储器中。

Spring 配置的事务传播属性如表 3-1 所示。

表 3-1 事务属性

传播行为	含 义
PROPAGATION_MANDATORY	表示该方法必须在事务中运行，如果当前事务不存在，则会抛出一个异常
PROPAGATION_NESTED	表示如果当前已经存在一个事务，那么该方法会在嵌套事务中运行。嵌套的事务可以独立于当前事务进行单独地提交或回滚。如果当前事务不存在，那么其行为与 PROPAGATION_REQUIRED 一样。注意各厂商对这种传播行为的支持是有所差异的。可以参考资源管理器的文档来确认它们是否支持嵌套事务
PROPAGATION_NEVER	表示当前方法不应该运行在事务上下文中。如果当前正有一个事务在运行，则会抛出异常

续表

传播行为	含 义
PROPAGATION_NOT_SUPPORTED	表示该方法不应该运行在事务中。如果存在当前事务，在该方法运行期间，当前事务将被挂起。如果使用 JTATransactionManager，则需要访问 TransactionManager
PROPAGATION_REQUIRED	表示当前方法必须运行在事务中。如果当前事务存在，则方法会在此事务中运行。否则，会启动一个新的事务
PROPAGATION_REQUIRED_NEW	表示当前方法必须运行在它自己的事务中。一个新的事务将被启动。如果存在当前事务，则在该方法执行期间，当前事务会被挂起。如果使用 JTATransactionManager，则需要访问 TransactionManager
PROPAGATION_SUPPORTS	表示当前方法不需要事务上下文，如果存在当前事务，那么该方法会在这个事务中运行

如果没有事务存在，则按照 REQUIRED 属性执行，该属性只对 DataSourceTransactionManager 事务管理器有效。

并发引起的事务问题

同一个应用程序或者不同应用程序中的多个事务在同一个数据集上并发执行时，可能会出现许多意外的问题。

并发事务所导致的问题可以分为以下三类。

- 脏读：脏读发生在一个事务读取了另一个事务改写但尚未提交的数据时。如果改写在稍后被回滚了，那么第一个事务获取的数据就是无效的。
- 不可重复读：不可重复读发生在一个事务执行相同的查询两次或两次以上，但是每次都得到不同的数据时。通常是因为另一个并发事务在两次查询期间更新了数据。
- 幻读：幻读与不可重复读类似。它发生在一个事务（T1）读取了几行数据，接着另一个并发事务（T2）插入了一些数据时。在随后的查询中，第一个事务（T1）就会发现多了一些原本不存在的记录。

Spring Boot 中的事务集成

Spring Boot 对事务的处理又进一步做了简化，首先在 pom.xml 文件中添加 Spring-boot-starter-jdbc 依赖，框架会默认注入 DataSourceTransactionManager 实例。不管是 JPA 还是 JDBC 等都实现自接口 PlatformTransactionManager。也可以添加 Spring-boot-starter-data-jpa 依赖，框架会默认注入 JpaTransactionManager 实例。这样就会在工程中默认加上事务，使用注解 @EnableTransactionManagement 开启事务支持后，然后在访问数据库的 Service 方法上添加注解

@Transactional 便可。

@Transactional 不仅可以注释在方法上，也可以注释在类上。当注释在类上时就意味着此类的所有 public 方法都是开启事务的。如果类级别和方法级别同时使用了 @Transactional 注释，则使用在类级别的注释会重载方法级别的注释。

事务的操作也比较简单。

实现

(1) 新建工程 chapter-03-04-04，并且新建表 User，设置 userName 的长度为 5，当超过这个长度时系统抛出异常，触发回滚操作。

加入正常数据测试：

```
public void testInsert() {
    for (int i = 0; i < 5; i++) {
        UserEntity userEntity = new UserEntity();
        userEntity.setPassWord("password "+i);
        userEntity.setUserName("i"+i);
        userEntity.setUserSex(UserSexEnum.MAN);
        userMapper.insert(userEntity);
    }
}
```

数据插入成功。

测试

启动应用，加入异常数据，并且添加事务。

```
@Transactional
public void testInsert() {
    for (int i = 0; i < 5; i++) {
        UserEntity userEntity = new UserEntity();
        userEntity.setPassWord("password "+i);
        userEntity.setUserName("i"+i);
        userEntity.setUserSex(UserSexEnum.MAN);
        userMapper.insert(userEntity);
    }
    UserEntity userEntity = new UserEntity();
    userEntity.setPassWord("password ");
```



```

        userEntity.setUserName("username");
        userEntity.setUserSex(UserSexEnum.MAN);
        userMapper.insert(userEntity);
    }

```

打开浏览器，输入测试地址 <http://localhost:8080/add>，可以看到控制台抛出异常

```

### Error updating database. Cause: com.mysql.jdbc.MysqlDataTruncation:
Data truncation: Data too long for column 'userName' at row 1
### The error may involve com.cloud.skyme.mapper.UserMapper.insert-Inline
### The error occurred while setting parameters
### SQL: INSERT INTO users(userName,passWord,user_sex) VALUES(?, ?, ?)
### Cause: com.mysql.jdbc.MysqlDataTruncation: Data truncation: Data too
long for column 'userName' at row 1

```

到库中查询，可以看到数据没有插入成功并且成功实现回滚。

3.4.5 整合 Redis

介绍

Redis 是目前业界使用最广泛的内存数据存储。相比 Memcached，Redis 支持更丰富的数据结构，比如 hashes、lists、sets 等，同时支持数据持久化。除此之外，Redis 还提供一些类数据库的特性，比如事务、HA、主从库。可以说 Redis 兼具了缓存系统和数据库的一些特性，因此有着丰富的应用场景。

Spring 定义了 `org.springframework.cache.CacheManager` 和 `org.springframework.cache.Cache` 接口来统一不同的缓存技术，而 Spring Boot 提供了自动配置多个 `CacheManager` 的实现。

接下来看一下缓存中比较重要的几个概念。

缓存命中率

- 即从缓存中读取数据的次数与总读取次数的比率，命中率越高越好；
- 命中率 = 从缓存中读取次数 / [总读取次数 (从缓存中读取次数 + 从慢速设备上读取的次数)]；
- Miss 率 = 没有从缓存中读取的次数 / [总读取次数 (从缓存中读取次数 + 从慢速设备上读取的次数)]。

一般通过这几个比率来监控缓存是否工作良好。

6 种数据淘汰策略

- **volatile-lru**: 从已设置过期时间的数据集 (`server.db[i].expires`) 中挑选最近最少使用的数据淘汰。
- **volatile-ttl**: 从已设置过期时间的数据集 (`server.db[i].expires`) 中挑选将要过期的数据淘汰。
- **volatile-random**: 从已设置过期时间的数据集 (`server.db[i].expires`) 中任意选择数据淘汰。
- **allkeys-lru**: 从数据集 (`server.db[i].dict`) 中挑选最近最少使用的数据淘汰。
- **allkeys-random**: 从数据集 (`server.db[i].dict`) 中任意选择数据淘汰。
- **no-eviction** (驱逐): 禁止驱逐数据。

策略使用规则:

- 如果数据呈现幂律分布, 也就是一部分数据访问频率高, 一部分数据访问频率低, 则使用 **allkeys-lru**。
- 如果数据呈现平等分布, 也就是所有的数据访问频率都相同, 则使用 **allkeys-random**。

Spring 提供了如下注解来声明缓存规则, 如表 3-2 所示。

表 3-2 Spring 注释声明缓存规则

注 解	描 述
@Cacheable	表明 Spring 在调用方法之前, 首先应该在缓存中查找方法的返回值。如果这个值能够找到, 则会返回缓存的值。否则, 这个方法就会被调用, 返回值会放到缓存中
@CachePut	表明 Spring 应该将方法的返回值放到缓存中。在方法的调用前并不会检查缓存, 方法始终都会被调用
@CacheEvict	表明 Spring 应该在缓存中清除一个或多个条目
@Caching	这是一个分组的注解, 能够同时应用多个其他的缓存注解
@CacheConfig	可以在类层级配置一些共用的缓存配置

Spring Boot 通过 `@enablecaching` 注解自动化配置合适的缓存管理器 (`CacheManager`), 默认情况下 Spring Boot 根据下面的顺序自动检测缓存提供者: `Generic`、`JCache` (`JSR-107`)、`EhCache 2.x`、`Hazelcast`、`Infinispan`、`Redis`、`Guava`、`Simple`。

实现

Spring Boot 整合 Redis 同样非常简单。

(1) 新建工程 `chapter-03-04-05`, 添加 Redis 的 Starter。

```
<dependency>
    <groupId>org.springframework.boot</groupId>
```



```
<artifactId>Spring-boot-starter-data-redis</artifactId>  
</dependency>
```

(2) 添加 Redis 相关的配置文件。

```
spring.redis.database=1  
spring.redis.host=127.0.0.1  
spring.redis.port=6379  
spring.redis.password=  
spring.redis.pool.max-idle=8  
spring.redis.pool.min-idle=0  
spring.redis.pool.max-active=8  
spring.redis.pool.max-wait=-1  
spring.redis.timeout=5000
```

(3) 添加类 RedisService, 引入 StringRedisTemplate。

```
@Service  
public class RedisService {  
  
    @Autowired  
    StringRedisTemplate stringRedisTemplate;  
  
    @Resource(name = "stringRedisTemplate")  
    ValueOperations<String, String> valOpsStr;  
  
    @Autowired  
    RedisTemplate<Object, Object> redisTemplate;  
  
    @Resource(name = "redisTemplate")  
    ValueOperations<Object, Object> valOpsObj;  
  
    /**  
     * 根据指定 key 获取 String  
     * @param key  
     * @return  
     */  
}
```

```

public String getStr(String key){
    return valOpsStr.get(key);
}

/**
 * 设置 Str 缓存
 * @param key
 * @param val
 */
public void setStr(String key, String val){
    valOpsStr.set(key, val);
}

/**
 * 删除指定 key
 * @param key
 */
public void del(String key){
    stringRedisTemplate.delete(key);
}

/**
 * 根据指定 o 获取 Object
 * @param o
 * @return
 */
public Object getObj(Object o){
    return valOpsObj.get(o);
}

/**
 * 设置 obj 缓存
 * @param o1
 * @param o2
 */
public void setObj(Object o1, Object o2){
    valOpsObj.set(o1, o2);
}

```



```

    /**
     * 删除 Obj 缓存
     * @param o
     */
    public void delObj(Object o){
        redisTemplate.delete(o);
    }
}

```

(4) 在 RedisController 类中添加测试代码。

测试

启动测试用的 Redis。

打开浏览器，输入 `http://127.0.0.1:8080/redis/setStr?key=aa&val=zhangfeng`。

设置完成后，数据就会存储到 Redis 中，然后使用 `http://127.0.0.1:8080/redis/getStr?key=aa`，查看设置的值是否能够获取，输出得到 `zhangfeng`，证明已经可以从缓存中获取值。然后删除缓存中的值：`http://127.0.0.1:8080/redis/delStr?key=aa`。

再次调用获取方法，则不能从缓存中得到信息。

测试缓存对象的流程与以上测试基本相同，输入的测试地址为：

`http://127.0.0.1:8080/redis/setObj?key=aa&id=1&name=zhangfeng`。

总结

Spring Boot 为我们整合 Redis 做了非常大的简化，在开发过程中，只需要简单的几项配置，就能够完成缓存的功能。

3.4.6 整合队列

介绍

消息队列中间件是分布式系统中重要的组件，主要解决应用耦合、异步消息、流量削峰等问题。实现高性能、高可用、可伸缩和最终一致性架构，是大型分布式系统不可缺少的中间件。

目前在生产环境中使用较多的消息队列有 ActiveMQ、RabbitMQ、ZeroMQ、Kafka、MetaMQ 和 RocketMQ 等。

AMQP 即 Advanced Message Queuing Protocol，是一个提供统一消息服务的应用层标准高级

消息队列协议，是应用层协议的一个开放标准，为面向消息的中间件设计。

Spring Boot 对消息队列的开发也做了简化，我们就用 RabbitMQ 作为示例，来说明整个整合过程。RabbitMQ 采用 Erlang 语言开发。Erlang 语言由 Ericson 设计，专门为开发 concurrent 和 distribution 系统的一种语言，在电信领域使用广泛。

RabbitMQ 的几个概念如下。

- **Broker**: 消息队列服务器实体。
- **Exchange**: 消息交换机，它指定消息按什么规则路由到哪个队列。
- **Queue**: 消息队列载体，每个消息都会被投入到一个或多个队列。
- **Binding**: 绑定，它的作用就是把 Exchange 和 Queue 按照路由规则绑定起来。
- **Routing Key**: 路由关键字，Exchange 根据这个关键字进行消息投递。
- **vhost**: 虚拟主机，一个 broker 里可以开设多个 vhost，用于不同用户的权限的分离。
- **producer**: 消息生产者，就是投递消息的程序。
- **consumer**: 消息消费者，就是接收消息的程序。
- **channel**: 消息通道，在客户端的每个连接里可以建立多个 channel，每个 channel 代表一个会话任务。

消息队列的使用过程如下。

- (1) 客户端连接到消息队列服务器，打开一个 channel。
- (2) 客户端声明一个 Exchange，并设置相关属性。
- (3) 客户端声明一个 Queue，并设置相关属性。
- (4) 客户端使用 Routing Key，在 Exchange 和 Queue 之间建立好绑定关系。
- (5) 客户端投递消息到 Exchange。

(6) Exchange 接收到消息后，就根据消息的 Key 和已经设置的 Binding，进行消息路由，将消息投递到一个或多个队列里。

RabbitMQ 的 Exchange 的四种类型分别为 Direct、topic、headers 和 Fanout。

- **Direct** 是 RabbitMQ 默认的交换机模式，也是最简单的模式，即创建消息队列时，指定一个 BindingKey。当发送者发送消息时，指定对应的 Key。当 Key 和消息队列的 BindingKey 一致的时候，消息将会被发送到该消息队列中。
- **topic** 转发信息主要是依据通配符，队列和交换机的绑定主要是依据一种模式（通配符 + 字符串），当发送消息时，只有指定的 Key 和该模式相匹配时，消息才会被发送到该

消息队列中。

- `headers` 也是根据一个规则进行匹配，在消息队列和交换机绑定时会指定一组键值对规则，而发送消息时也会指定一组键值对规则，当两组键值对规则相匹配时，消息会被发送到匹配的消息队列中。
- `Fanout` 是路由广播的形式，会把消息发给绑定它的全部队列，即便设置了 `Key`，也会被忽略。

实现

消息队列收发的测试流程如下。

- (1) 安装 RabbitMQ。
- (2) 新建工程 `chapter-03-04-06`，并且加入 `Spring-boot-starter-amqp` 的 Starter。

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>Spring-boot-starter-amqp</artifactId>
</dependency>
```

- (3) 在配置文件中加入配置。

```
Spring.application.name=chapter-03-04-06
Spring.rabbitmq.host=127.0.0.1
Spring.rabbitmq.port=5672
Spring.rabbitmq.username=guest
Spring.rabbitmq.password=guest
Spring.rabbitmq.publisher-confirms=true
Spring.rabbitmq.virtual-host=
```

- (4) 使用 `Direct` 模式，指定队列名。Rabbitmq 既支持字符串的发送，也支持序列化的方式。

```
final static String queueName = "hello";//指定队列名
@RabbitListener(queues = "hello")//消息监听方法上加入监听的队列名
```

- (5) 单个生产者、多个消费者的情况，只需要在接收方法上指定要监听的队列名就可以实现消费。

- (6) `ExChange` 方式，`topic` 是 RabbitMQ 中最灵活的一种方式，可以根据 `binding_key` 自由地绑定不同的队列。

```
@Bean
public Queue queueMessage() {
    return new Queue("topic.message");
}

@Bean
public Queue queueMessages() {
    return new Queue("topic.messages");
}

@Bean
public TopicExchange exchange() {
    return new TopicExchange("exchange");
}

@Bean
public FanoutExchange fanoutExchange() {
    return new FanoutExchange("fanoutExchange");
}

@Bean
public Binding bindingExchangeMessage(Queue queueMessage, TopicExchange
exchange) {
    return BindingBuilder.bind(queueMessage).to(exchange).with("topic.message");
}

@Bean
public Binding bindingExchangeMessages(Queue queueMessages, TopicExchange
exchange) {
    return BindingBuilder.bind(queueMessages).to(exchange).with("topic.#");
}
```

(7) fanout ExChange 方式，也就是我们常说的广播或者订阅发布模式，所有订阅的接收都能够消费发送的消息。

```
@Bean
public Queue AMessage() {
    return new Queue("fanout.A");
}
```



```

    }

    @Bean
    public Queue BMessage() {
        return new Queue("fanout.B");
    }

    @Bean
    public Queue CMessage() {
        return new Queue("fanout.C");
    }

    @Bean
    FanoutExchange fanoutExchange() {
        return new FanoutExchange("fanoutExchange");
    }

    @Bean
    Binding bindingExchangeA(Queue AMessage, FanoutExchange fanoutExchange) {
        return BindingBuilder.bind(AMessage).to(fanoutExchange);
    }

    @Bean
    Binding bindingExchangeB(Queue BMessage, FanoutExchange fanoutExchange) {
        return BindingBuilder.bind(BMessage).to(fanoutExchange);
    }

    @Bean
    Binding bindingExchangeC(Queue CMessage, FanoutExchange fanoutExchange) {
        return BindingBuilder.bind(CMessage).to(fanoutExchange);
    }

```

测试

(1) 测试 Direct 模式，启动应用，打开浏览器并且输入 <http://localhost:8080/rabbit/send>，可以看到消息发送成功并且消息成功被消费。

```

2018-03-07 11:50:03.238 INFO 1404 --- [nio-8080-exec-1]
c.c.skyme.controller.RabbitController : 消息发送

```

```
2018-03-07 11:50:03.238 INFO 1404 --- [nio-8080-exec-1]
com.cloud.skyme.queue.HelloSender : 消息发送 : hello Wed Mar 07
11:50:03 CST 2018
2018-03-07 11:50:03.602 INFO 1404 --- [cTaskExecutor-1]
com.cloud.skyme.queue.HelloReceiver : 接收到的消息 : User [userName=张锋,
password=123456, age=30]
```

Direct 模式相当于一对一模式，一个消息被发送者发送后，会被转发到一个绑定的消息队列中，然后被一个接收者接收。

(2) 单生产多消费的情况测试，打开浏览器输入 <http://localhost:8080/rabbit/sendMany>，可以看到 10 个队列分别被不同的消费者消费。

```
2018-03-07 11:51:50.708 INFO 1404 --- [nio-8080-exec-8]
c.c.skyme.controller.RabbitController : 消息发送
2018-03-07 11:51:50.708 INFO 1404 --- [nio-8080-exec-8]
com.cloud.skyme.queue.HelloSender : 消息发送 : hello Wed Mar 07 11:51:50
CST 2018
2018-03-07 11:51:50.709 INFO 1404 --- [nio-8080-exec-8]
com.cloud.skyme.queue.HelloSender : 消息发送 : hello Wed Mar 07 11:51:50
CST 2018
2018-03-07 11:51:50.710 INFO 1404 --- [nio-8080-exec-8]
com.cloud.skyme.queue.HelloSender : 消息发送 : hello Wed Mar 07 11:51:50
CST 2018
2018-03-07 11:51:50.710 INFO 1404 --- [nio-8080-exec-8]
com.cloud.skyme.queue.HelloSender : 消息发送 : hello Wed Mar 07 11:51:50
CST 2018
2018-03-07 11:51:50.710 INFO 1404 --- [nio-8080-exec-8]
com.cloud.skyme.queue.HelloSender : 消息发送 : hello Wed Mar 07 11:51:50
CST 2018
2018-03-07 11:51:50.711 INFO 1404 --- [nio-8080-exec-8]
com.cloud.skyme.queue.HelloSender : 消息发送 : hello Wed Mar 07 11:51:50
CST 2018
2018-03-07 11:51:50.711 INFO 1404 --- [nio-8080-exec-8]
com.cloud.skyme.queue.HelloSender : 消息发送 : hello Wed Mar 07 11:51:50
CST 2018
2018-03-07 11:51:50.711 INFO 1404 --- [nio-8080-exec-8]
com.cloud.skyme.queue.HelloSender : 消息发送 : hello Wed Mar 07 11:51:50
CST 2018
```



```

2018-03-07 11:51:50.712 INFO 1404 --- [nio-8080-exec-8]
com.cloud.skyme.queue.HelloSender      : 消息发送 : hello Wed Mar 07 11:51:50
CST 2018

2018-03-07 11:51:50.712 INFO 1404 --- [nio-8080-exec-8]
com.cloud.skyme.queue.HelloSender      : 消息发送 : hello Wed Mar 07 11:51:50
CST 2018

```

由以上结果可知,生产者发送的 10 条消息分别被两个消费者接收了。此种情况既可以是单生产对应多个消费,也可以扩展成多生产、多消费的情况。

(3) topic 消息测试,打开浏览器输入 <http://localhost:8080/rabbit/topic>。

```

2018-03-07 11:52:47.263 INFO 1404 --- [nio-8080-exec-2]
com.cloud.skyme.queue.TopicSender      : 发送者 1 : I am topic.mesaage msg=====
2018-03-07 11:52:47.264 INFO 1404 --- [nio-8080-exec-2]
com.cloud.skyme.queue.TopicSender      : 发送者 2 : I am topic.mesaages
msg#####

2018-03-07 11:52:47.270 INFO 1404 --- [cTaskExecutor-1]
com.cloud.skyme.queue.Receiver1         : receiver1 接收到的消息 : I am
topic.mesaage msg=====

2018-03-07 11:52:47.280 INFO 1404 --- [cTaskExecutor-1]
com.cloud.skyme.queue.Receiver1         : receiver2 接收到的消息 : I am
topic.mesaage msg=====

2018-03-07 11:52:47.281 INFO 1404 --- [cTaskExecutor-1]
com.cloud.skyme.queue.Receiver1         : receiver2 接收到的消息 : I am
topic.mesaages msg#####

```

由以上结果可知: sender1 发送的消息, routing_key 是 “topic.message”, Exchange 里面的绑定的 binding_key 是 “topic.message”, topic.# 都符合路由规则,所以 sender1 发送的消息,两个队列都能接收到。

sender2 发送的消息, routing_key 是 “topic.messages”, Exchange 里面的绑定的 binding_key 只有 topic.# 符合路由规则,所以 sender2 发送的消息只有队列 topic.messages 能收到。

(4) 测试 fanout exchange 类型 RabbitMQ, 打开浏览器输入 <http://localhost:8080/rabbit/fanoutSender>。

```

2018-03-07 11:53:41.715 INFO 1404 --- [nio-8080-exec-5]
com.cloud.skyme.queue.TopicSender      : fanoutSender :hello i am zhangfeng
2018-03-07 11:53:41.721 INFO 1404 --- [cTaskExecutor-1]

```

```
com.cloud.skyme.queue.FanoutReceiverA : FanoutReceiverA :
fanoutSender :hello i am zhangfeng
    2018-03-07 11:53:41.721 INFO 1404 --- [cTaskExecutor-1]
com.cloud.skyme.queue.FanoutReceiverB : FanoutReceiverB :
fanoutSender :hello i am zhangfeng
    2018-03-07 11:53:41.721 INFO 1404 --- [cTaskExecutor-1]
com.cloud.skyme.queue.FanoutReceiverB : FanoutReceiverC :
fanoutSender :hello i am zhangfeng
```

由以上结果可知：fanoutSender 发送消息时指定了 routing_key 为“abcd.ee”，所有接收者都接收到了消息。

总结

消息队列的使用在企业中很常见，经常用于跨语言的数据传输，RabbitMQ 应对大数据量的传输有着得天独厚的优势，并且可以扩展成集群，承担更大的压力。

3.4.7 操作 MongoDB

介绍

MongoDB 是一个基于分布式文件存储的数据库，由 C++ 语言编写，旨在为 Web 应用提供可扩展的高性能数据存储解决方案。

MongoDB 是一个介于关系数据库和非关系数据库之间的产品，是非关系数据库中功能最丰富、最像关系数据库的。它支持的数据结构非常松散，是类似 JSON 的 BSON 格式，因此可以存储比较复杂的数据类型。MongoDB 最大的特点是支持的查询语言非常强大，其语法有点类似于面向对象的查询语言，几乎可以实现类似关系数据库单表查询的绝大部分功能，而且还支持对数据建立索引。

实现

- (1) 安装 MongoDB 并新建工程 chapter-03-04-07。
- (2) 添加 Spring-boot-starter-data-mongodb 的 Starter。

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>Spring-boot-starter-data-mongodb</artifactId>
</dependency>
```


(3) 添加配置, 连接 MongoDB。

```
Spring.data.mongodb.uri=mongodb://localhost:27017/test
```

(4) 注入 MongoTemplate, 完成基础操作。

```
@Autowired
private MongoTemplate mongoTemplate;
```

测试

在 JUnit 中测试对 MongoDB 的 CRUD 操作。

注入 UserDao, 并且运行 CRUD 的单元测试, 可以在 MongoDB 的客户端查看运行结果。

```
@Autowired
private UserDao userDao;
```

分别运行测试并且查看查询结果, 可以得到如图 3-10 所示的结果。

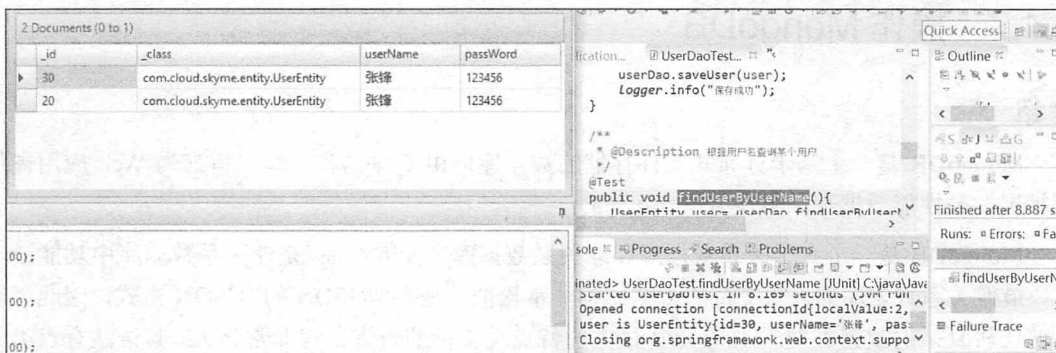


图 3-10 操作 MongoDB

将 MongoDB 中结果为张锋的数据查出, 也可以查询多条记录, 这里只演示查询一条数据的情况。

总结

MongoDB 是最类似关系型数据库的 NoSQL 数据库, 对于互联网及应用经常变化的业务支持得特别好。在企业中有着非常广泛的应用, 为我们应对复杂变化的业务提供了一种非常好的选择。

Spring Boot 对于数据的持久化提供了非常丰富的整合和简化, 这也是微服务提倡的, 对于某些服务可能使用 MongoDB 更合适, 比如业务中遇到的数据经常变化的情况, 有些情况使用

关系型数据库更合适。而对于高并发访问的情况，直接暴露数据库给用户访问往往会对数据库造成非常大的压力，所以这个时候把数据持久化到 Redis 中是一种非常好的选择。总之一句话，适合的就是最好的。

3.5 Web 开发

介绍

Web 开发是现在企业应用中非常重要的一环，在以前的开发方式中，我们需要编写大量的配置文件，整理非常多的 XML 配置，如果出现问题，经常需要调试很长时间。Spring Boot 也对 Web 开发进行了进一步的简化，Spring Boot 提供了 Spring-boot-starter-web 来为 Web 开发予以支持，Spring-boot-starter-web 为我们提供了嵌入的 Tomcat 和 SpringMVC 的依赖。同时，也提供了对大量模板引擎的整合，包括 FreeMarker、Groovy、Thymeleaf、Velocity 和 Mustache 等，官方推荐的是 Thymeleaf。

Thymeleaf 是一个 XML/XHTML/HTML5 模板引擎，可以用于 Web 与非 Web 应用。

Thymeleaf 的主要目标在于提供一种可被浏览器正确显示的、格式良好的模板创建方式，因此也可以用作静态建模。可以使用它创建经过验证的 XML 与 HTML 模板。相对于编写逻辑或代码，开发者只需将标签属性添加到模板中即可。接下来，这些标签属性就会在 DOM（文档对象模型）上执行预先制定好的逻辑。Thymeleaf 的可扩展性也非常好，可以使用它定义自己的模板属性集合，这样就可以计算自定义表达式并使用自定义逻辑，这意味着 Thymeleaf 还可以作为模板引擎框架。

在 Spring Boot 中，对资源文件的路径做了约定，资源文件的约定目录结构如下。

- Maven 的资源文件目录：/src/java/resources;
- Spring-boot 项目静态文件目录：/src/java/resources/static;
- Spring-boot 项目模板文件目录：/src/java/resources/templates;
- Spring-boot 静态首页的支持，即 index.html 放在以下目录结构会直接映射到应用的根目录下。

```
classpath:/META-INF/resources/index.html
classpath:/resources/index.html
classpath:/static/index.html
classpath:/public/index.html
```

在 Spring-boot 下，默认约定了 Controller 视图跳转中 Thymeleaf 模板文件的前缀 prefix 是“classpath:/templates/”，后缀 suffix 是“.html”。

这个在 `application.properties` 配置文件中是可以修改的。

如下配置可以修改视图跳转的前缀和后缀：

```
Spring.thymeleaf.prefix: /templates/
Spring.thymeleaf.suffix: .html
```

更多有关 Thymeleaf 中的默认配置可以查看 `org.springframework.boot.autoconfigure.thymeleaf.ThymeleafProperties` 类的属性。

如果我们需要在一台计算机上启动多个应用，就会遇到端口冲突的问题，可以在配置文件中直接修改 Tomcat 启动的端口，来实现自定义启动端口。这个操作也非常简单，直接修改端口配置：

```
server.port=8088
```

实现

(1) 新建工程 `chapter-03-05`，加入 Web 支持，因为我们要整合 Thymeleaf，所以需要加入 Thymeleaf 的 Starter。

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>Spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

(2) 开发过程中需要关闭 Thymeleaf 缓存，只有部署到线上才能够打开，不然有可能造成修改无效的情况出现。在配置文件中加入 Thymeleaf 配置，如果不修改可使用默认配置。

```
#####
###THYMELEAF 自动配置
#####
#Spring.thymeleaf.prefix=classpath:/templates/
#Spring.thymeleaf.suffix=.html
#Spring.thymeleaf.mode=HTML5
#Spring.thymeleaf.encoding=UTF-8
# ;charset=<encoding> is added
#Spring.thymeleaf.content-type=text/html
# set to false for hot refresh
#指定关闭缓存
Spring.thymeleaf.cache=false
```


这里我们指定路径为 `classpath:/templates/`，也就是工程中对应的 `src/main/resources/templates` 目录。

(3) 编写测试模板，有关 Thymeleaf 的语法请自行查找，在模板中指定变量 `hello`，此部分会接收后端传来的参数进行动态显示。

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org"
      xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-Springsecurity3">
  <head>
    <title>Hello World!</title>
  </head>
  <body>
    <h1 th:inline="text">Hello.Spring Boot</h1>
    <p th:text="${hello}"></p>
  </body>
</html>
```

(4) 编写后端测试类 `WebController`，传入参数，并且在前台页面展示。

```
@RequestMapping("/thymeleaf")
public String helloThymeleaf(Map<String, Object> map) {
    map.put("hello", "从模板后台传入并显示");
    return "/helloSpringBoot";
}
```

(5) Spring Boot 除了支持 Thymeleaf，同样也支持 Freemarker 等模板引擎，并且能够在同一工程中使用多个模板引擎。配置 Freemarker 等与配置 Thymeleaf 基本相同。

(6) 在 Web 开发的过程中，使用自定义拦截器是常见的场景。如果想要创建自定义拦截器，需要创建自己的拦截器类并实现 `HandlerInterceptor` 接口，然后创建一个 Java 类继承 `WebMvcConfigurerAdapter`，并重写 `addInterceptors` 方法。最后实例化自定义的拦截器，将对象手动添加到拦截器链中（在 `addInterceptors` 方法中添加）。自定义拦截器的代码对应本书附带的源代码 `MyInterceptorPermissions`、`MyInterceptorVerification` 和 `MyWebAppConfigurer` 类，内容可以根据自己的业务自行扩展。

测试

(1) Web 访问测试，启动应用，打开浏览器输入 `http://localhost:8080/thymeleaf`，我们就可

以看到后台传入的参数，如图 3-11 所示。



图 3-11 后台传入参数显示

(2) 测试自定义拦截器。这里需要注意一下，如果上边修改端口，那么打开浏览器后输入的地址就需要换成新的端口，输入地址为 `http://localhost:8088/thymeleaf`，我们能够看到在控制台上打印出自定义拦截器的内容。

```
2017-11-13 11:05:09.715 INFO 4372 --- [nio-8088-exec-1]
c.c.s.i.MyInterceptorVerification : >>>MyInterceptorVerification>>>>>>
在请求处理之前进行调用 (Controller 方法调用之前)
2017-11-13 11:05:09.716 INFO 4372 --- [nio-8088-exec-1]
c.c.s.i.MyInterceptorPermissions : >>>权限验证>>>>>>在请求处理之前进行调用
(Controller 方法调用之前)
2017-11-13 11:05:09.732 INFO 4372 --- [nio-8088-exec-1]
c.c.s.i.MyInterceptorPermissions : >>>权限验证请求处理之后进行调用，但是在视
图被渲染之前 (Controller 方法调用之后)
2017-11-13 11:05:09.732 INFO 4372 --- [nio-8088-exec-1]
c.c.s.i.MyInterceptorVerification : >>>MyInterceptorVerification>>>>>>
请求处理之后进行调用，但是在视图被渲染之前 (Controller 方法调用之后)
2017-11-13 11:05:10.050 INFO 4372 --- [nio-8088-exec-1]
c.c.s.i.MyInterceptorPermissions : >>>权限验证在整个请求结束之后被调用，也就
是在 DispatcherServlet 渲染了对应的视图之后执行 (主要用于进行资源清理工作)
2017-11-13 11:05:10.050 INFO 4372 --- [nio-8088-exec-1]
c.c.s.i.MyInterceptorVerification : >>>MyInterceptorVerification>>>>>>
在整个请求结束之后被调用，也就是在 DispatcherServlet 渲染了对应的视图之后执行 (主要用于进
行资源清理工作)
```

总结

Web 开发是企业级开发中非常重要的一环，系统从原来的 C/S 架构逐渐演化为 B/S 架构，并且演变到现在的微服务架构，Web 开发都起着非常重要的作用。像现在的互联网网站，展示到用户面前的都是网页或者 H5 的页面，Spring Boot 为我们做 Web 开发自动封装了一套非常灵

活的机制，只需要掌握，就能够加速开发，起到事半功倍的效果。

3.6 懒人的接口文档管理

介绍

在接触的很多开发人员中，对于文档的编写都非常抵触，如果不是公司强制执行，一般都是草草了事。

而 Spring Boot 为我们提供了非常好的接口文档管理插件 Swagger，可以说是完美集成，公司领导再也不用为开发人员不提交更新文档而苦恼了。

Swagger 是一个规范和完整的框架，用于生成、描述、调用和可视化 RESTful 风格的 Web 服务。总体目标是使客户端和文件系统作为服务器以同样的速度来更新。文件的方法、参数和模型紧密集成到服务器端的代码，允许 API 来始终保持同步。

Swagger 相关的注解如下。

- `@Api`: 修饰整个类，描述 Controller 的作用。
- `@ApiOperation`: 描述一个类的一个方法，或者说一个接口。
- `@ApiParam`: 单个参数描述。
- `@ApiModel`: 用对象来接收参数。
- `@ApiModelProperty`: 用对象接收参数时，描述对象的一个字段。
- `@ApiResponse`: HTTP 响应其中 1 个描述。
- `@ApiResponses`: HTTP 响应整体描述。
- `@ApiIgnore`: 使用该注解忽略这个 API。
- `@ApiError`: 发生错误返回的信息。
- `@ApiImplicitParam`: 一个请求参数。
- `@ApiImplicitParams`: 多个请求参数。

实现

(1) 新建工程 chapter-03-06，加入 Web 支持，并且在 pom.xml 中添加整合 Swagger2，添加依赖。

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>Springfox-swagger2</artifactId>
```



```

        <version>2.7.0</version>
    </dependency>

    <dependency>
        <groupId>io.Springfox</groupId>
        <artifactId>Springfox-swagger-ui</artifactId>
        <version>2.7.0</version>
    </dependency>

```

(2) 指定配置和格式，添加类 `Swagger2Config`。

```

@Configuration
@EnableSwagger2
public class Swagger2Config {

    @Bean
    public Docket createRestApi() {
        return new Docket(DocumentationType.SWAGGER_2)
            .apiInfo(apiInfo())
            .select()
            .apis(RequestHandlerSelectors.basePackage("com.cloud.skyme"))
            .paths(PathSelectors.any())
            .build();
    }

    private ApiInfo apiInfo() {
        return new ApiInfoBuilder()
            .title("Springboot 利用 Swagger 构建 API 文档")
            .description("简单优雅的 restfun 风格, http://www.cnblogs.com/skyme/")
            .termsOfServiceUrl("http://www.cnblogs.com/skyme/")
            .version("1.0")
            .build();
    }
}

```

`@Configuration` 注解表明它是一个配置类，`@EnableSwagger2` 开启 `Swagger2`，`apiInfo()` 配

置一些基本的信息，`apis()`指定扫描的包会生成文档。

(3) 编写接口类，并且添加上相应的标签。

```
@RestController
@RequestMapping("/user")
@Api("UserController 相关 api")
public class UserController {
    @ApiOperation("获取用户信息")
    @ApiImplicitParams({
        @ApiImplicitParam(paramType = "header", name = "username", dataType
= "String", required = true, value = "用户的姓名", defaultValue = "zhangfeng"),
        @ApiImplicitParam(paramType = "query", name = "password", dataType
= "String", required = true, value = "用户的密码", defaultValue = "123456") })
    @ApiResponses({ @ApiResponse(code = 400, message = "请求参数没填好"),
        @ApiResponse(code = 404, message = "请求路径没有或页面跳转路径不对") })
    @RequestMapping(value = "/getUser", method = RequestMethod.GET)
    public String getUser(@RequestHeader("username") String username,
        @RequestParam("password") String password) {
        return username + password + "接口测试";
    }
}
```

相关注解的解释如下。

- **@Api**: 用在类上，说明该类的作用。
- **@ApiOperation**: 用在方法上，说明方法的作用。
- **@ApiImplicitParams**: 用在方法上，包含一组参数说明。
- **@ApiImplicitParam**: 用在**@ApiImplicitParams**注解中，指定一个请求参数的各个方面。
 - **paramType**: 参数放在哪个地方。
 - ✓ **header**-->请求参数的获取: **@RequestHeader**。
 - ✓ **query**-->请求参数的获取: **@RequestParam**。
 - ✓ **path** (用于 RESTful 接口) -->请求参数的获取: **@PathVariable**。
 - **name**: 参数名。
 - **dataType**: 参数类型。
 - **required**: 参数是否必须传。
 - **value**: 参数的意思。

- `defaultValue`: 参数的默认值。
- `@ApiResponses`: 用于表示一组响应。
- `@ApiResponse`: 用在`@ApiResponses`中，一般用于表达一个错误的响应信息。
 - `code`: 数字，例如，400。
 - `message`: 信息，例如，“请求参数输入有误”。
 - `response`: 抛出异常的类。
- `@ApiModel`: 描述一个 Model 的信息（这种一般用在 post 创建时，使用`@RequestBody`的场景，请求参数无法使用`@ApiImplicitParam`注解进行描述时）。
 - `@ApiModelProperty`: 描述一个 model 的属性。

测试

工程构建完成后，启动应用，打开浏览器并且输入 `http://localhost:8080/swagger-ui.html`，可以看到如图 3-12 所示的界面。

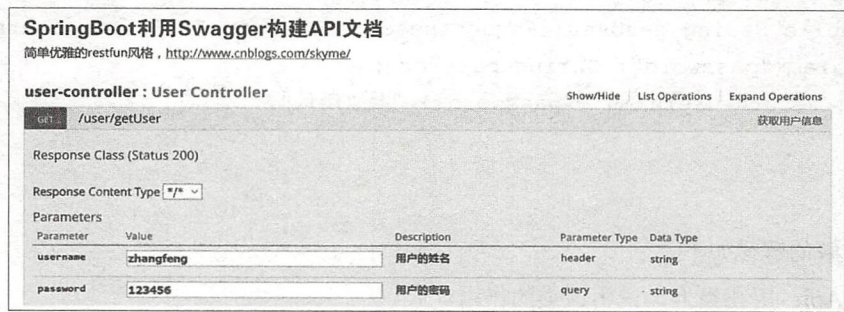


图 3-12 Swagger 接口文档

输入相应的参数并单击“Try it out!”，就可以提交请求并且测试接口是否正常。

总结

在实际的业务场景中，一般通过 Swagger2 对接口的文档进行维护，然后将各个微服务所提供的接口文档地址统一维护到 wiki 中，这样整个团队就有了统一的标准接口文档，并且能够完成基本的测试功能，实在是程序员的一大福音。

3.7 优化的调度

介绍

定时任务对于每个人来说都不陌生，在企业级应用中也普遍使用，比如我们想从某个接口

定期地获取指定的数据，就需要配置定时任务，可以使用 timer 方式，也可以使用 cron 表达式的方式进行配置。Spring Boot 提供了非常简便的配置调度的方式，严格来说，这个应该属于 Spring 4 提供的。在企业级开发中，一般不推荐把定时任务散布在各个工程中，而是应该以集中的方式进行管理，在后面的章节中，会重点提及分布式调度这一话题。

定时任务的实现方式：

- Java 自带的 `java.util.Timer` 类，这个类允许你调度一个 `java.util.TimerTask` 任务。使用这种方式可以让程序按照某一个频度执行，但不能在指定时间运行。
- 使用 Quartz，这是一个功能比较强大的调度器，可以让程序在指定时间执行，也可以按照某一个频度执行，配置起来稍显复杂。
- Spring Boot 自带的 `Scheduled`，可以将它看作一个轻量级的 Quartz，而且使用起来比 Quartz 简单许多。

@Scheduled 注释：

- `fixdDelay`——以指定时间间隔调度（以方法执行结束时间为准）。
- `fixedRate`——以指定时间间隔调度任务（以方法执行开始时间为准）。
- `initialDelay`——指定延迟后开始调度任务。
- `cron 表达式`——使用 cron 表达式来配置调度。

实现

(1) 新建工程，添加调度类代码，在需要调度的方法上边加上 @Scheduled，并且设定好参数。

```
@Scheduled(fixedRate = 5000)
```

(2) 在应用启动类上添加 @EnableScheduling，使应用支持调度。

(3) 其实现方式与其他工程类似，只是加上注解的方式和参数不同。

测试

启动应用，并且监控控制台，可以看到如下输出：

```
2017-11-13 11:07:05.001 INFO 22300 --- [pool-1-thread-1]
com.cloud.skyme.schedule.ScheduledTasks : The time is now 11:07:05
2017-11-13 11:07:10.002 INFO 22300 --- [pool-1-thread-1]
com.cloud.skyme.schedule.ScheduledTasks : The time is now 11:07:10
2017-11-13 11:07:15.000 INFO 22300 --- [pool-1-thread-1]
com.cloud.skyme.schedule.ScheduledTasks : The time is now 11:07:15
```

```
2017-11-13 11:07:20.001 INFO 22300 --- [pool-1-thread-1]
com.cloud.skyme.schedule.ScheduledTasks : The time is now 11:07:20
2017-11-13 11:07:25.001 INFO 22300 --- [pool-1-thread-1]
com.cloud.skyme.schedule.ScheduledTasks : The time is now 11:07:25
```

5 秒执行一次任务。

总结

通过 Spring Boot 提供的优化的调度方式，在一般的企业级应用中，可以非常轻松地应对各种情况的调度。但是对于比较大型的应用来说，这种方式未免不够灵活，并且不易于管理，比如想在控制台批量开启或者批量停止一批业务，处理起来就会非常吃力，而这也正是需要分布式调度的原因。

3.8 健康是永恒的主题

介绍

赚钱虽然重要，但是身体更重要！伟人教导我们，身体是革命的本钱。失去了健康，就失去了一切。

对于系统来说，也是一样的道理，不能够持续提供服务，就算有再多功能也等于 0。Spring Boot 的 actuator 提供的对应用系统的自省和监控的集成功能，可以对应用系统进行配置查看、相关功能统计等操作。

实现

新建工程并加依赖：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>Spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>Spring-boot-starter-web</artifactId>
</dependency>
```

添加后，当应用启动时，就可以通过它提供的接口检查应用的健康情况。主要功能如表 3-3 所示。

表 3-3 Spring Boot 健康检查

HTTP 方法	路 径	描 述	鉴 权
GET	/autoconfig	查看自动配置的使用情况	true
GET	/configprops	查看配置属性, 包括默认配置	true
GET	/beans	查看 Bean 及其关系列表	true
GET	/dump	打印线程栈	true
GET	/env	查看所有环境变量	true
GET	/env/{name}	查看具体变量值	true
GET	/health	查看应用健康指标	false
GET	/info	查看应用信息	false
GET	/mappings	查看所有 URL 映射	true
GET	/metrics	查看应用基本指标	true
GET	/metrics/{name}	查看具体指标	true
POST	/shutdown	关闭应用	true
GET	/trace	查看基本追踪信息	true

测试

启动应用, 并且查看控制台, 能够看到如下输出:

```
2018-03-05 19:02:00.223 INFO 6428 --- [           main] s.b.a.e.w.s.
WebMvcEndpointHandlerMapping : Mapped "{[/actuator/health],methods=[GET],
produces=[application/vnd.spring-boot.actuator.v2+json || application/json]}"
onto public java.lang.Object org.springframework.boot.actuate.endpoint.web.
servlet.AbstractWebMvcEndpointHandlerMapping$OperationHandler.handle(javax.
servlet.http.HttpServletRequest,java.util.Map<java.lang.String,
java.lang.String>)
```

```
2018-03-05 19:02:00.225 INFO 6428 --- [           main] s.b.a.e.w.s.
WebMvcEndpointHandlerMapping : Mapped "{[/actuator/info],methods=[GET],
produces=[application/vnd.spring-boot.actuator.v2+json || application/json]}"
onto public java.lang.Object org.springframework.boot.actuate.endpoint.web.
servlet.AbstractWebMvcEndpointHandlerMapping$OperationHandler.handle(javax.
servlet.http.HttpServletRequest,java.util.Map<java.lang.String, java.lang.String>)
```

```
2018-03-05 19:02:00.226 INFO 6428 --- [           main] s.b.a.e.w.s.
WebMvcEndpointHandlerMapping : Mapped "{[/actuator],methods=[GET],produces=
[application/vnd.spring-boot.actuator.v2+json || application/json]}" onto
protected java.util.Map<java.lang.String, java.util.Map<java.lang.String,
org.springframework.boot.actuate.endpoint.web.Link>> org.springframework.boot.
```

```
actuate.endpoint.web.servlet.WebMvcEndpointHandlerMapping.links(javax.servlet.  
http.HttpServletRequest,javax.servlet.http.HttpServletResponse)  
2018-03-05 19:02:00.358 INFO 6428 --- [           main] o.s.j.e.a.  
AnnotationMBeanExporter      : Registering beans for JMX exposure on startup  
2018-03-05 19:02:00.436 INFO 6428 --- [           main] o.s.b.w.embedded.  
tomcat.TomcatWebServer      : Tomcat started on port(s): 8080 (http) with context  
path ''
```

现在就可以通过提供的接口服务查看应用的健康情况及 JVM 的使用情况等信息了。

总结

健康检查是微服务中非常重要的一个环节，通过收集这些信息，能够知道硬件和应用的健康情况，及时做出调整，为用户提供更好的体验，保证应用的健壮性。

3.9 强强联合之整合 Dubbo

介绍

Dubbo 是分布式系统比较受欢迎的一套框架，也是很多公司从单体应用转到分布式应用的不二选择。而且 Dubbo 也可以跟第三方的框架整合，形成非常强大的服务架构。Spring Boot 为 Dubbo 的使用提供了一套非常方便的接口，使大家无须配置繁杂的 XML 文件。

使用 Dubbo 构建的微服务，最重要的是提供了调度中间层，整体架构如图 3-13 所示。

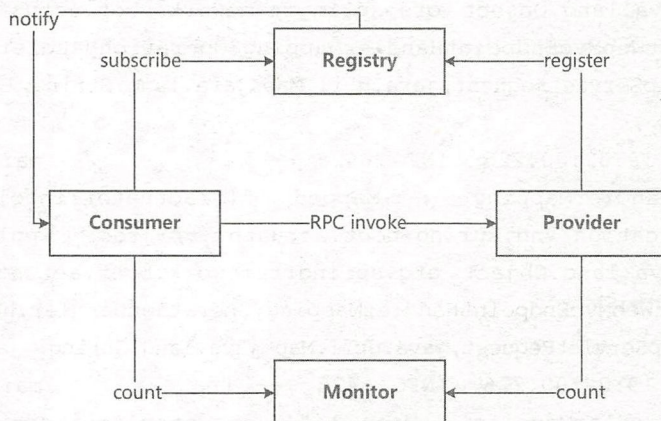


图 3-13 Dubbo 整体架构

- 调用中间层变成了可选组件，消费者可以直接访问服务提供者。

- 服务信息被集中到 Registry 中，形成了服务治理的中心组件。
- 通过 Monitor 监控系统，可以直观地展示服务调用的统计信息。
- Consumer 可以进行负载均衡、服务降级的选择。

实现

(1) 安装 ZooKeeper。

(2) 因为是分布式的场景，所以需要两个工程，一个是服务的提供者工程 chapter-03-09-provider，一个是服务的调用者工程 chapter-03-09-customer。

(3) 在提供者项目中添加 DubboConfig 类，用于读取配置文件中的 Dubbo 配置，然后在配置文件中添加 Dubbo 相关的配置，指定注册中心为 ZooKeeper。配置文件中的内容如下：

```
# Dubbo 配置，指定扫描的包和刷新周期
Spring.dubbo.privider.scan.package=com.cloud.skyme
Spring.dubbo.provider.config.timeout=10000
Spring.dubbo.provider.config.retries=1

Spring.dubbo.application.config.name=provider

Spring.dubbo.registry.config.protocol=zookeeper
Spring.dubbo.monitor.protocol=registry

Spring.dubbo.protocol.config.name=dubbo
Spring.dubbo.protocol.config.port=20880
Spring.dubbo.registry.config.address=localhost:2181
```

(4) 添加服务接口和服务实现，注意服务实现上的注释引用需要使用 Dubbo 提供的注释 com.alibaba.dubbo.config.annotation.Service，否则消费端不能够找到要消费的服务，需要添加的代码如下：

```
@Service
@Component
public class UserServiceImpl implements UserService
```

(5) 在消费端同样加入 DubboConfig 类，用于加载 Dubbo 配置，因为消费端为调用层，可以直接使用 Spring Boot 提供的 DubboConfig 类，并且加入之前讲到的 Swagger2Config 类的配置。消费端的配置信息如下：

```
# Dubbo 配置, 指定扫描的包和刷新周期
Spring.dubbo.provider.scan.package=com.cloud.skyme
Spring.dubbo.provider.config.timeout=10000
Spring.dubbo.provider.config.retries=1

Spring.dubbo.application.config.name=customer

Spring.dubbo.registry.config.protocol=zookeeper
Spring.dubbo.monitor.protocol=registry

server.port=8088
Spring.dubbo.registry.config.address=localhost:2181
```

这里需要将注册中心与服务端指向同一个注册中心, 不然无法消费提供者提供的服务。

(6) 新建 `UserController` 类, 用于消费服务端的服务, 因为是通过接口方式的调用, 所以消费端需要有服务端的接口声明, 并且引用时使用 `com.alibaba.dubbo.config.annotation.Reference` 来消费服务。调用提供者服务的代码如下:

```
@Reference
private UserService userService;
```

测试

(1) 启动本地安装的 ZooKeeper。

(2) 启动服务提供者, 能够在控制台看到输出。

```
2018-03-05 19:13:14.984 INFO 13980 --- [          main]
s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped
"[/addUser,methods=[POST]]" onto public java.lang.String
com.cloud.skyme.controller.UserController.addUser()

2018-03-05 19:13:14.986 INFO 13980 --- [          main]
s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped
"[/getUser,methods=[POST]]" onto public java.lang.String
com.cloud.skyme.controller.UserController.getUser()

2018-03-05 19:13:14.996 INFO 13980 --- [          main]
s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped
"[/swagger-resources/configuration/ui]" onto public
org.springframework.http.ResponseEntity<springfox.documentation.swagger.web.
UiConfiguration>
```



```

springfox.documentation.swagger.web.ApiResourceController.uiConfiguration()
    2018-03-05 19:13:14.997 INFO 13980 --- [main]
s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/swagger-resources]}"
onto public
org.springframework.http.ResponseEntity<java.util.List<springfox.documentati
on.swagger.web.SwaggerResource>>
springfox.documentation.swagger.web.ApiResourceController.swaggerResources()
    2018-03-05 19:13:14.998 INFO 13980 --- [main]
s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped
"{[/swagger-resources/configuration/security]}" onto public
org.springframework.http.ResponseEntity<springfox.documentation.swagger.web.
SecurityConfiguration>
springfox.documentation.swagger.web.ApiResourceController.securityConfigurat
ion()
    2018-03-05 19:13:15.008 INFO 13980 --- [main]
s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/error]}" onto public
org.springframework.http.ResponseEntity<java.util.Map<java.lang.String,
java.lang.Object>>
org.springframework.boot.autoconfigure.web.servlet.error.BasicErrorControlle
r.error(javax.servlet.http.HttpServletRequest)
    2018-03-05 19:13:15.009 INFO 13980 --- [main]
s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped
"{[/error],produces=[text/html]}" onto public
org.springframework.web.servlet.ModelAndView
org.springframework.boot.autoconfigure.web.servlet.error.BasicErrorControlle
r.errorHtml(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpSer
vletResponse)
    2018-03-05 19:13:15.357 INFO 13980 --- [main]
pertySourcedRequestMappingHandlerMapping : Mapped URL path [/v2/api-docs] onto
method [public
org.springframework.http.ResponseEntity<springfox.documentation.spring.web.j
son.Json>
springfox.documentation.swagger2.web.Swagger2Controller.getDocumentation(jav
a.lang.String,javax.servlet.http.HttpServletRequest)]
    2018-03-05 19:13:15.828 INFO 13980 --- [main]
s.w.s.m.m.a.RequestMappingHandlerAdapter : Looking for @ControllerAdvice:
org.springframework.boot.web.servlet.context.AnnotationConfigServletWebServe
rApplicationContext@2fd6b6c7: startup date [Mon Mar 05 19:13:10 CST 2018]; root

```

```

of context hierarchy
2018-03-05 19:13:15.904 INFO 13980 --- [           main]
o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/webjars/**] onto
handler of type [class
org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
2018-03-05 19:13:15.904 INFO 13980 --- [           main]
o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/**] onto handler
of type [class
org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
2018-03-05 19:13:15.956 INFO 13980 --- [           main]
o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/**/favicon.ico]
onto handler of type [class
org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
2018-03-05 19:13:16.190 INFO 13980 --- [           main]
o.s.j.e.a.AnnotationMBeanExporter      : Registering beans for JMX exposure on
startup
2018-03-05 19:13:16.203 INFO 13980 --- [           main]
o.s.c.support.DefaultLifecycleProcessor : Starting beans in phase 2147483647
2018-03-05 19:13:16.204 INFO 13980 --- [           main]
d.s.w.p.DocumentationPluginsBootstrapper : Context refreshed
2018-03-05 19:13:16.233 INFO 13980 --- [           main]
d.s.w.p.DocumentationPluginsBootstrapper : Found 1 custom documentation
plugin(s)
2018-03-05 19:13:16.305 INFO 13980 --- [           main]
s.d.s.w.s.ApiListingReferenceScanner    : Scanning for api listing references
2018-03-05 19:13:16.484 INFO 13980 --- [           main]
o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8088
(http) with context path ''

```

(3) 启动消费者，然后打开浏览器，输入 <http://localhost:8088/swagger-ui.html>，得到测试界面，然后分别测试添加和获取用户接口。

(4) 调用添加用户方法，可以先看到消费端控制台输入消费 `addUser` 方法，然后提供者输出添加用户 `zhangfeng`，并且能够在界面上看到返回。

添加用户 zhangfeng 成功

(5) 调用获取用户方法，可以看到添加了同样的效果，并且能够看到返回。

获取用户 zhangfeng 成功

方法调用完成后，查看控制台，能够看到服务提供者输出如下内容：

```
2018-03-05 19:14:13.289 INFO 13980 --- [nio-8088-exec-4]
c.cloud.skyme.controller.UserController : 消费 addUser 方法
2018-03-05 19:14:57.024 INFO 13980 --- [nio-8088-exec-5]
c.cloud.skyme.controller.UserController : 消费 getUser 方法
```

总结

Dubbo 类似一个组装机，可以同任何不同的框架进行整合，这是它的优点，同样也是它的缺点，也就是说，整合的时候会遇到各种各样的问题。比如：

- Registry 严重依赖第三方组件（ZooKeeper 或者 Redis），当这些组件出现问题时，服务调用很快就会中断。
- Dubbo 只支持 RPC 调用，使得服务提供方与调用方在代码上产生了强依赖，服务提供者需要不断将包含公共代码的 JAR 包打包出来供消费者使用。一旦打包出现问题，就会导致服务调用出错。

3.10 小结

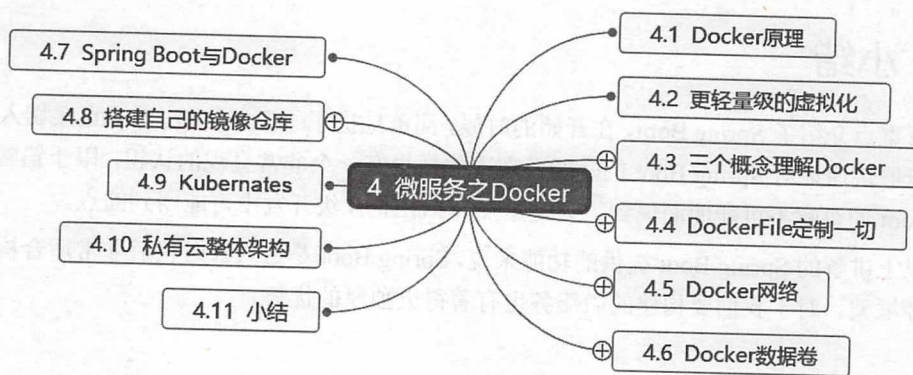
本章重点介绍了 Spring Boot，在开始的时候，简单地提了一下 Spring，然后直接切入主题，让读者能够快速地对 Spring Boot 的功能及实现的效果有一个非常直观的认识，限于篇幅，对于 Spring Boot 的讲解不可能面面俱到，只是重点介绍在企业级开发中可能用到的点。

从以上讲解的 Spring Boot 提供的功能来说，Spring Boot 是进入云时代后非常适合构建云服务的一种框架。对于我们要构建的微服务也有着得天独厚的优势。



4 chapter

第 4 章 微服务之 Docker



Docker 是个伟大的项目，它彻底释放了虚拟化的威力，让应用的分发、部署和管理都变得前所未有的高效和轻松！

如果你正在为以下问题困扰，就可以考虑使用 Docker 来对整个工程进行重构。

➤ 资源利用率问题

不同业务场景对资源的需求是不一样的，有 CPU 密集型、内存密集型、网络密集型，这就可能导致资源利用率不合理的问题。在大多数的企业中，我们经常能够看到服务器的资源闲置率非常高，而开发团队却经常面临无机器可用的情况。

➤ 混合部署交叉影响

对于线上服务，一台机器要混合部署多个服务，那么服务之间可能存在相互影响的情况。比如端口的冲突，CPU 及内存的共同使用等问题，都有可能造成服务之间的冲突，从而导致莫名的问题。

➤ 扩/缩容效率低

当业务节点需要进行扩/缩容时，从机器下线到应用部署、测试，周期较长。当业务遇到突发流量高峰时，增加设备并部署后，可能流量高峰已经过去了。这些本身与设计无关，即使接口应用已经设计成无状态的，但是想要做扩容也是一件非常麻烦的事。

➤ 多环境代码不一致

大部分公司都会有由于过去内部开发流程的不规范而存在一些问题，业务提测的代码在测试环境测试完毕后，在线上部署时可能会进行修改、调整，然后打包上线。这就会导致测试的代码和线上运行的代码是不一致的，增加了服务上线的风险，也增加了线上服务故障排查的难度。

➤ 缺少稳定的线下测试环境

在测试过程中，会遇到一个问题，服务依赖的其他下游服务都没有提供稳定的测试环境，导致无法在测试环境模拟整个线上流程进行测试。所以有时候会用线上服务进行测试，这本身就存在着很高的潜在风险。一旦操作失误，就有可能造成不可估量的损失。

而 Docker 的出现在很大程度上解决了这些问题，那么我们一起来认识一下 Docker 吧。

Docker 是一个开源的引擎，可以轻松地为任何应用创建一个轻量级的、可移植的、自给自足的容器。开发者在笔记本电脑上编译测试通过的容器可以批量地在生产环境中部署，包括 VMs（虚拟机）、bare metal、OpenStack 集群和其他基础应用平台。

Docker 的目标：

- 提供轻量简单的建模方式；
- 职责的逻辑分离；



- 快速高效的开发生命周期;
- 鼓励使用面向服务的架构, 即单个容器运行单个应用。

4.1 Docker 原理

Docker 是一个客户端—服务器 (C/S) 架构的程序。Docker 客户端只需向 Docker 服务器或守护进程发出请求, 服务器或守护进程将完成所有工作并返回结果。Docker 提供了一个命令行工具 Docker 及一整套 RESTful API, 可以在同一台宿主机上运行 Docker 守护进程和客户端, 也可以从本地的 Docker 客户端连接到运行在另一台宿主机上的远程 Docker 守护进程。图 4-1 描述了 Docker 的运行原理。

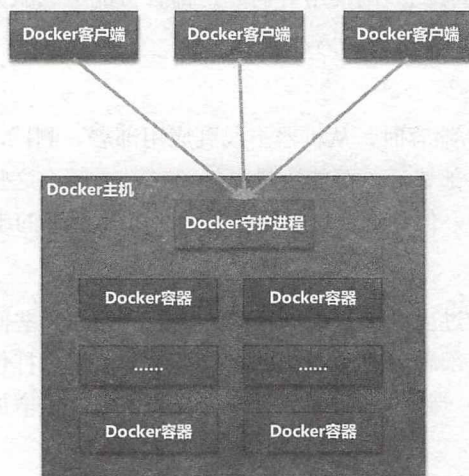


图 4-1 Docker 的运行原理

Docker 依赖的 Linux 的内核特性包括 Namespaces 命名空间和 Control groups(c groups)控制组。

Namespaces 命名空间:

- PID (process ID), 进程 ID 隔离;
- NET (network), 管理网络端口;
- IPC (InterProcess Communication), 进程间通信;
- 管理跨进程通信的访问;
- MNT (Mount), 管理挂载点;
- UTS (UNIX Timesharing System), 隔离内核和版本标识;



- Control groups(c groups), 控制组。

Control groups(c groups)控制组:

- 资源限制;
- 优先级设定;
- 资源度量;
- 资源控制及资源分配;

Docker 容器的能力包括:

- 文件系统隔离——每个容器都有自己的 root 文件系统, 可以独立挂载外部文件系统。
- 进程隔离——每个容器都运行在自己的进程环境中, 相互之间互不干扰。
- 网络隔离——容器间的虚拟网络接口和 IP 地址都是分开的。
- 资源隔离和分组——使用 c group 将 CPU 和内存等资源独立分配给每个 Docker 容器。

4.2 更轻量级的虚拟化

Docker 项目的目标是实现轻量级的操作系统虚拟化解方案。我们先来看一下虚拟机与 Docker 的架构对比, 如图 4-2 所示。

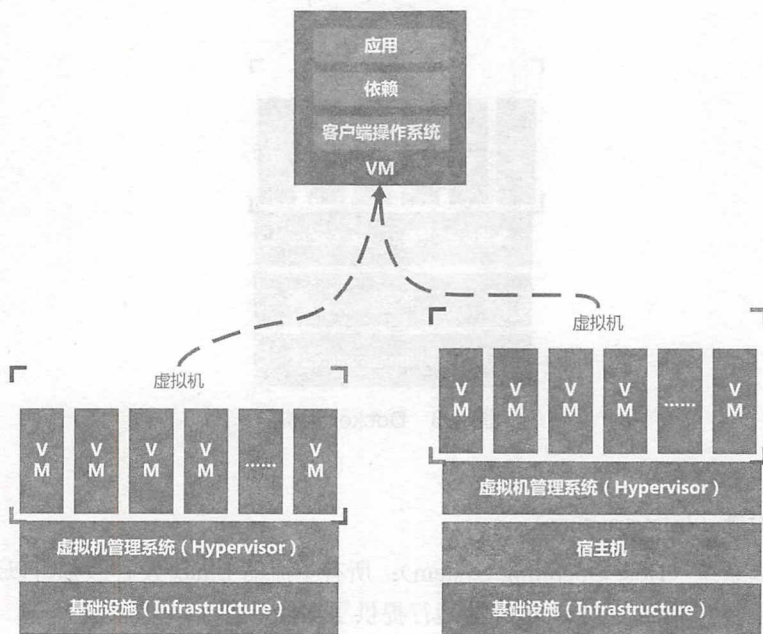


图 4-2 Docker 和虚拟化对比



从下到上理解图 4-2。

- 基础设施 (Infrastructure): 一般是服务器或者云主机。
- 虚拟机管理系统 (Hypervisor): 利用 Hypervisor, 可以在主操作系统之上运行多个不同的从操作系统, 可以构建在基础设施上, 也可以构建在操作系统上。
- 客户机操作系统 (Guest Operating System): 假设运行 3 个相互隔离的应用, 则需要使用 Hypervisor 启动 3 个客户机操作系统, 也就是 3 个虚拟机。这些虚拟机都非常大, 也许有 900MB, 这就意味着它们将占用 2.7GB 的磁盘空间。更糟糕的是, 它们还会消耗很多 CPU 和内存资源。
- 各种依赖: 每一个客户机操作系统都需要安装许多依赖。
- 应用: 安装依赖之后, 就可以在各个客户机操作系统分别运行应用了, 这样各个应用就是相互隔离的。

再来看一下 Docker 的架构, 如图 4-3 所示。

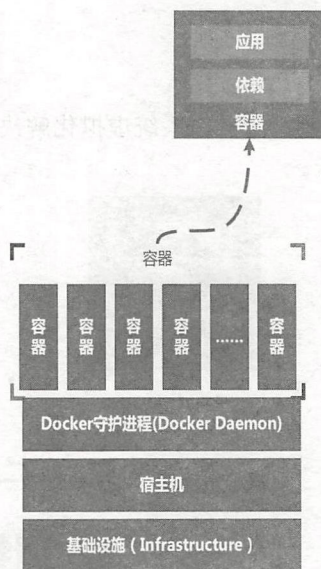


图 4-3 Docker 架构

从下到上理解图 4-3。

- 基础设施 (Infrastructure)。
- 主操作系统 (Host Operating System): 所有主流的 Linux 发行版都可以运行 Docker。Windows 和 Mac 也对 Docker 的运行提供了支持。
- Docker 守护进程 (Docker Daemon): Docker 守护进程取代了 Hypervisor, 它是运行在

操作系统之上的后台进程，负责管理 Docker 容器。

- 各种依赖：对于 Docker，应用的所有依赖都打包在 Docker 镜像中，Docker 容器是基于 Docker 镜像创建的。
- 应用：应用的源代码与它的依赖都打包在 Docker 镜像中，不同的应用需要不同的 Docker 镜像。不同的应用运行在不同的 Docker 容器中，它们是相互隔离的。

虚拟机和 Docker 的对比：

- Docker 容器可以在秒级实现，比虚拟机的方式不只快了一倍，任何虚拟机都不太可能在秒级启动完成。
- Docker 对系统资源的利用率很高，一台主机上可以同时运行数千个 Docker 容器。如果把服务器比作码头，则虚拟机就好比码头上的仓库，不能随便移动，而 Docker 就好比集装箱，操作灵活，运载方便。
- 容器除了运行其中应用，基本不消耗额外的系统资源，所使用的资源完全是使用宿主主机上的资源。
- 传统虚拟机方式运行 10 个不同的应用就要启动 10 个虚拟机，而 Docker 只需要 10 个隔离的应用即可。
- Docker 容器可以跨平台运行，不需要额外的操作系统支持，按需装载。

4.3 三个概念理解 Docker

Docker 有 3 个基本概念，镜像、容器和仓库。可以说理解了这几个概念，就掌握了容器使用的基础知识。

在这之前，我们需要安装 Docker 环境，操作系统选择 CentOS 7，安装过程如下：

- (1) 切换到 root 权限。
- (2) 执行 `yum install Docker`。
- (3) 启动 Docker 并将 Docker 设置为开机启动。

```
systemctl start docker.service
systemctl enable docker.service
```

这样我们就能够执行基本的 Docker 命令了。如果想同时管理多个容器，需要安装 Docker Compose。安装过程如下。

进入网站获取最新版本：<https://github.com/docker/compose/releases>。

使用如下命令：

```
curl -L https://github.com/docker/compose/releases/download/1.17.1/  
docker-compose-`uname -s`-`uname -m` -o /usr/local/bin/docker-compose  
chmod +x /usr/local/bin/docker-compose
```

若/usr/bin 下找不到命令，则在 cp /usr/local/bin/docker-compose /usr/bin/ 查找下即可。最后执行 docker-compose -version 查看安装情况。

接下来看一下镜像、容器和镜像仓库之间的关系。

如图 4-4 所示，我们可以从镜像仓库摘取镜像到本地，然后将镜像启动，这样一个容器就构建完成了。Docker 之所以这么吸引人，除了它的新颖的技术，围绕官方 Registry (Docker Hub) 的生态圈也是相当吸引人眼球的地方。在 Docker Hub 上可以很容易下载到大量已经容器化好的应用镜像，即拉即用。这些镜像中，有些是 Docker 官方维护的，但更多的是众多开发者分享上传的。

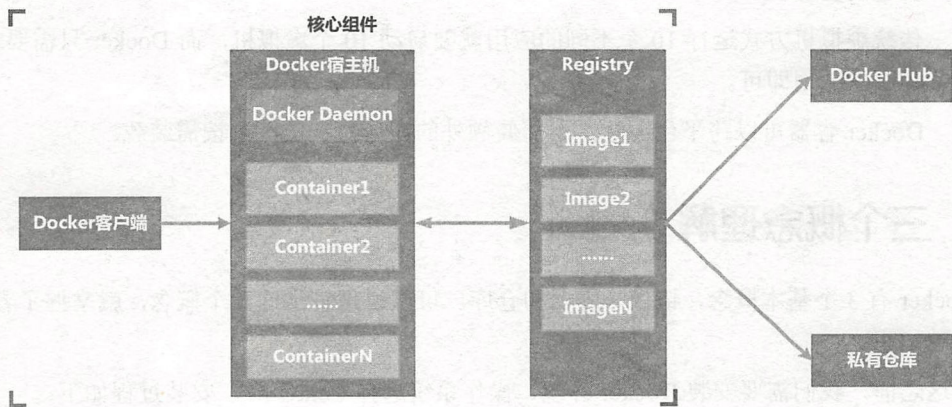


图 4-4 容器和镜像仓库之间的关系

4.3.1 镜像 (Image)

Docker Image 是需要定制化 Build 的一个“安装包”，包括基础镜像+应用的二进制部署包。这是一个没有运行的容器，或者说是没有启动的应用。类似一个模板，基于这个模板，可以构建出我们需要的环境。比如我们需要一个 Java 的运行环境，那么这个镜像中就需要有基础操作系统、JDK 的环境，如果是 Web 应用，就需要有 Tomcat 的支持，当然也可以直接运行 Spring Boot，用它内嵌的 Tomcat。

而镜像是分层的，也就是说，我们可以按照需求向上叠加。

(1) 搜索镜像，一般的学习都是从 hello-world 开始的，我们也不例外。执行：

```
docker search hello-world
```

可以看到列表中列出的所有镜像：

NAME	DESCRIPTION	STARS	OFFICIAL	AUTOMATED
hello-world	Hello World! (an example of minimal Docker...	453	[OK]	
kitematic/hello-world-nginx	A light-weight nginx container that demons...	95		
tutum/hello-world	Image to test docker deployments. Has Apac...	48	[OK]	
dockercloud/hello-world	Hello World!	13	[OK]	

我们选择获得星级最高的下载。

(2) 运行 docker pull，指定镜像名称。

```
docker pull hello-world
```

在下载过程中会输出每一层的信息。有时因为国内的一些限制，访问国外的资源会比较慢，所以可以指定国内的镜像仓库下载镜像。

(3) 查看所有下载的镜像，Docker Images（列出本地主机已有的所有镜像）。

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
jhipster/jhipster	master	52cbc76f5571	2 months ago	1.06GB
jhipster/jhipster	<none>	5ce5edb02dff	4 months ago	1.06GB
centos	7	5a962dd35afd	5 months ago	326MB
jhipster/jhipster	latest	955ea47ebc89	5 months ago	1.05GB
qlwl-rule-engine	1.0.0	78c10911ae90	5 months ago	929MB
centos	<none>	196e0ce0c9fb	5 months ago	197MB
zookeeper	latest	f2249a75c5d0	5 months ago	143MB
hello-world	latest	1815c82652c0	8 months ago	1.84kB
vmware/harbor-log	v1.1.1	9c46a7b5e517	12 months ago	192MB
vmware/harbor-log	v1.1.2	9c46a7b5e517	12 months ago	192MB
jplock/zookeeper	3.4.8	80e2f8f23445	24 months ago	155MB

在列出信息中，可以看到几个字段信息：

- REPOSITORY 表示来自哪个仓库库，比如 centos；
- TAG 是镜像的标记，比如 7，通常用来区分仓库中的多个镜像，如果不指定，默认就是 latest；

- IMAGE ID 是它的 ID 号（唯一）；
- CREATED 是创建时间；
- SIZE 是镜像大小。

（4）删除镜像，使用 `docker rmi` 加镜像名，这里需要注意，有该镜像创建的容器存在时，镜像文件默认是无法删除的，所以删除镜像前最好删除所有依赖该镜像的容器，不要用强制删除。

（5）创建镜像有 3 种方法：

- 基于已有镜像的容器创建时使用 `docker commit`；
- 基于本地模板导入；
- 基于 Dockerfile 创建。

（6）其他的操作镜像的命令还包括：保存镜像使用 `doker save`；载入镜像使用 `docker load`；上传镜像使用 `docker push` 等。

4.3.2 容器 (Container)

Docker Container 是 Image 的实例，可以运行不同操作系统的 Image。Docker 利用容器来运行应用，容器是从镜像创建的运行实例，它可以被启动、开始、停止、删除。可以把容器看作一个最小化版的 Linux。并且每个容器都是相互隔离的，可以在其中运行自己需要的环境而不影响其他环境。

（1）运行上面下载的镜像 `hello-word`，使用 `docker run` 创建一个新的容器并运行一个命令。

```
docker run hello-world
```

可以看到运行的结果：

```
[root@pek-docker-03 ~]# docker run hello-world
```

```
Hello from Docker!
```

```
This message shows that your installation appears to be working correctly.
```

命令格式：`docker run [OPTIONS] IMAGE [COMMAND] [ARG...]`。

参数说明：

- `-a stdin`——指定标准输入输出内容类型，可选 `STDIN/STDOUT/STDERR` 三项；

- `-d`——后台运行容器，并返回容器 ID；
- `-i`——以交互模式运行容器，通常与 `-t` 同时使用；
- `-t`——为容器重新分配一个伪输入终端，通常与 `-i` 同时使用；
- `--name="nginx-lb"`——为容器指定一个名称；
- `--dns 8.8.8.8`——指定容器使用的 DNS 服务器，默认和宿主一致；
- `--dns-search example.com`——指定容器 DNS 搜索域名，默认和宿主一致；
- `-h "mars"`——指定容器的 hostname；
- `-e username="ritchie"`——设置环境变量；
- `--env-file=[]`——从指定文件读入环境变量；
- `--cpuset="0-2" or --cpuset="0,1,2"`——绑定容器到指定 CPU 运行；
- `-m`——设置容器使用内存最大值；
- `--net="bridge"`——指定容器的网络连接类型，支持 bridge/host/none/container: 四种类型；
- `--link=[]`——添加链接到另一个容器；
- `--expose=[]`——开放一个端口或一组端口。

(2) 进入容器的几种方法。

➤ 使用 docker attach

Docker 提供了 `attach` 命令来进入 Docker 容器，只需要加上要进入的容器 ID 就可以进入容器了。但是使用该命令有一个问题，当多个窗口同时使用该命令进入该容器时，所有的窗口都会同步显示。如果有一个窗口阻塞了，那么其他窗口也无法进行操作。所以，这样的命令是无法用在生产环境中的。

➤ 使用 SSH

看到这个会想到虚拟机，也就是说，如果想通过这种方式进入，首先要在容器中安装 SSH Server，然后通过 SSH 以客户端的方式连接，这种方式虽然能够解决问题，但是对于使用来说太重了，也就是人们常说的“大炮打蚊子”，大材小用了。

➤ 使用 exec

Docker 在 1.3.X 版本之后还提供了一个新的命令 `exec` 用于进入容器，先执行 `Docker ps` 获取容器的 ID，然后执行 `Docker exec -it {容器的 ID} /bin/bash`，这种方式虽然方便，但是有版本的限制。

➤ 使用 nsenter

`nsenter` 是一个小工具，它可以进入命名空间，或者在现有的命名空间中产生一个新的进程。

系统一般会为我们默认安装 `nsenter`，如果没有安装就手动执行一下安装命令：

```
wget https://www.kernel.org/pub/linux/utils/util-linux/v2.24/util-linux-2.24.tar.gz
tar -xzf util-linux-2.24.tar.gz
cd util-linux-2.24/
./configure --without-ncurses
make nsenter
```

安装好后，我们需要使用 `docker inspect` 命令来获取运行的容器的进程 ID，先执行 `docker ps` 获取想要进入的容器的 CONTAINER ID，然后执行 `docker inspect {容器 ID}`，一般会输出很多信息，所以截取其中需要的部分，执行 `docker inspect -f {{.State.Pid}} {容器 ID}`，这样就可以获取运行的进程的 PID 了。然后执行 `nsenter --target {运行容器的 PID} --mount --uts --ipc --net --pid`，就可以进入容器中了，如果嫌这些操作麻烦，可以使用网上已经封装好的脚本进入。

(3) 运行 `docker ps` 可以看到所有正在运行的容器的状态。

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS		NAMES
2eb7ef8b2082	jhipster/jhipster	"tail -f /home/jhi..."	2 months ago
Up 2 months	0.0.0.0:3001->3001/tcp, 0.0.0.0:9000->9000/tcp, 0.0.0.0:8090->8080/tcp	jhipster	

常用的监控命令包括：

```
docker ps -a -q | wc -l    监控容器数量
docker ps -q | wc -l      正在运行的容器的数量
docker ps -a | grep -v 'Up ' | grep -v 'CONTAINER' | wc -l 非运行状态的容器的数量
```

(4) 启动守护式容器：守护式容器（`daemonized container`）没有交互式会话，适合运行应用程序和服务。守护容器可以添加 `-d` 参数来实现。如果想查看日志中的输出，使用 `docker logs` 获取容器运行时的输出日志，可以加上 `-tail` 参数设置要显示的条数。

(5) 其他的容器操作命令还包括：创建容器使用 `docker create`；终止容器使用 `docker stop`；启动容器使用 `docker start`；重启容器使用 `docker restart`；删除容器使用 `docker rm`；导出容器使用 `docker export` 等。

4.3.3 仓库 (Repository)

仓库是存放镜像的地方，仓库又可以分为公有仓库和私有仓库。

当用户创建了自己的镜像之后就可以使用 `push` 命令将它上传到公有或者私有仓库，这样下次在另外一台机器上使用这个镜像时，只需要从仓库上“pull”下来就可以了。

(1) 仓库登录使用 `docker login`。

(2) 管理私有仓库。

- 假设私有仓库地址为 192.168.0.100，端口为 5000；
- `docker tag ubuntu:14.04 192.168.0.100:5000/test`（更改标签名）；
- `docker push 192.168.0.100:5000/test`（push 镜像，会自动“push”到 192.168.0.100 地址的机器上）；
- `curl http://192.168.0.100:5000/v1/search`（查看仓库 192.168.0.100:5000 中是否含有 test 镜像）；
- `docker pull 192.168.0.100:5000/test`（在任何一台能访问到 192.168.0.100 地址的机器上下载镜像）。

如果想自己搭建仓库，推荐使用 Harbor，这是一款非常专业的私有仓库管理软件。详细的安装使用可参考“搭建自己的镜像仓库”部分。

4.4 Dockerfile 定制一切

为什么要用定制呢？Docker 为我们提供了一套非常强大的语法，可以利用简单的编写程序构建出任何你想要的环境，同时可以跟业务代码相结合，快速构建和生成所需要的应用。

Dockerfile 用来创建一个自定义的 Image，包含了用户指定的软件依赖等。使用 Docker 的 `build` 命令可以直接构建新的 Image。它简化了从头到尾的流程并极大地简化了部署工作。

4.4.1 Dockerfile 语法

Dockerfile 语法由两部分构成：注释和命令+参数。

```
# Line blocks used for commenting  
command argument argument ..
```

一个简单的示例：

```
# Print "Hello docker!"  
RUN echo "Hello docker!"
```

用这个简单的语法就可以构建出一个 Docker 镜像。

4.4.2 Dockerfile 命令

Dockerfile 的指令如下。

➤ FROM

FROM 命令可能是最重要的 Dockerfile 命令。此命令定义了使用哪个基础镜像启动构建流程。基础镜像可以为任意镜像。如果基础镜像没有被发现，则 Docker 将试图从 Docker image index 来查找该镜像。

格式为 FROM <image>或 FROM <image>:<tag>。

第一条指令必须为 FROM 指令。并且，在同一个 Dockerfile 中创建多个镜像时，可以使用多个 FROM 指令（每个镜像一次）。

➤ MAINTAINER

格式为 MAINTAINER <name>, 指定维护者信息。其实就是类似 javadoc 注释中的 @author, 用于声明创作者，一般都放在文件比较靠上的位置。

➤ RUN

从名称上就可以看出，RUN 是执行或运行的意思。

格式为 RUN <command>或 RUN ["executable", "param1", "param2"]。

前者将在 Shell 终端中运行命令，即/bin/sh -c；后者则使用 exec 执行。指定使用其他终端可以通过第二种方式实现，例如，RUN ["/bin/bash", "-c", "echo hello"]。

每条 RUN 指令将在当前镜像基础上执行指定命令，并提交为新的镜像。当命令较长时可以使用 “\” 来换行。

➤ EXPOSE

格式为 EXPOSE <port> [<port>...]

告诉 Docker 服务端容器暴露的端口号，供互联系统使用。

➤ CMD

支持三种格式：

- CMD ["executable", "param1", "param2"]使用 exec 执行，推荐方式；

- CMD command param1 param2 在/bin/sh 中执行，提供给需要交互的应用；
- CMD ["param1","param2"]提供给 ENTRYPOINT 的默认参数。

指定启动容器时执行的命令，每个 Dockerfile 只能有一条 CMD 命令。如果指定了多条命令，则只有最后一条会被执行。

如果用户启动容器时指定了运行的命令，则会覆盖 CMD 指定的命令。

➤ ENTRYPOINT

两种格式：

- ENTRYPOINT ["executable", "param1", "param2"];
- ENTRYPOINT command param1 param2 (Shell 中执行)。

配置容器启动后执行的命令，并且不可被 Docker RUN 提供的参数覆盖。

每个 Dockerfile 中只能有一个 ENTRYPOINT，当指定多个时，只有最后一个生效。

➤ ENV

格式为 ENV <key> <value>。指定一个环境变量，会被后续 RUN 指令使用，并在容器运行时保持。

例如：

```
ENV PG_MAJOR 9.3
ENV PG_VERSION 9.3.4
RUN curl -SL http://example.com/postgres-$PG_VERSION.tar.xz | tar -xJC
/usr/src/postgress && ...
ENV PATH /usr/local/postgres-$PG_MAJOR/bin:$PATH
```

➤ ADD

格式为 ADD <src> <dest>。

该命令将复制指定的<src>到容器中的<dest>，其中<src>可以是 Dockerfile 所在目录的一个相对路径；也可以是一个 URL；还可以是一个 tar 文件（自动解压为目录）。

➤ COPY

格式为 COPY <src> <dest>。

复制本地主机的<src>（Dockerfile 所在目录的相对路径）到容器中的<dest>。

当使用本地目录为源目录时，推荐使用 COPY。

➤ VOLUME

格式为 VOLUME ["/data"]。

创建一个可以从本地主机或其他容器挂载的挂载点，一般用来存放数据库和需要保持的数据等。

➤ WORKDIR

格式为 `WORKDIR /path/to/workdir`。

为后续的 `RUN`、`CMD`、`ENTRYPOINT` 指令配置工作目录。

可以使用多个 `WORKDIR` 指令，后续命令如果参数是相对路径，则会基于之前命令指定的路径。例如：

```
WORKDIR /a
WORKDIR b
WORKDIR c
RUN pwd
```

最终路径为 `/a/b/c`。

➤ USER

格式为 `USER daemon`。

指定运行容器时的用户名或 `UID`，后续的 `RUN` 也会使用指定用户。

当服务不需要管理员权限时，可以通过该命令指定运行用户。并且可以在之前创建所需要的用户，例如，`RUN groupadd -r postgres && useradd -r -g postgres postgres`。要临时获取管理员权限可以使用 `gosu`，而不推荐使用 `sudo`。

➤ ONBUILD

格式为 `ONBUILD [INSTRUCTION]`。

配置当创建的镜像作为其他新创建镜像的基础镜像时，所执行的操作指令。

例如，`Dockerfile` 使用如下内容创建了镜像 `image-A`。

```
[...]
ONBUILD ADD . /app/src
ONBUILD RUN /usr/local/bin/python-build --dir /app/src
[...]
```

如果基于 `A` 创建新的镜像，新的 `Dockerfile` 中使用 `FROM image-A` 指定基础镜像，则会自动执行 `ONBUILD` 指令内容，等价于在后面添加了两条指令。

```
FROM image-A
```



```
#Automatically run the following
ADD . /app/src
RUN /usr/local/bin/python-build --dir /app/src
```

使用 ONBUILD 指令的镜像，推荐在标签中注明，例如，ruby:1.9-onbuild。

4.4.3 Dockerfile 构建过程

Dockerfile 其实可以看作一个命令集，每行均为一条命令，每行的第一个单词就是命令 command，后面的字符串是该命令所要接收的参数。比如 ENTRYPOINT /bin/bash。ENTRYPOINT 命令的作用就是将后面的参数设置为镜像的 entrypoint。

➤ docker build 的流程

(1) 提取 Dockerfile (evaluator.go/RUN)。

(2) 将 Dockerfile 按行进行分析 (parser/parser.go/Parse)，每行第一个单词 (如 CMD、FROM 等) 叫作 command。根据 command，将之后的字符串用对应的数据结构进行接收。

(3) 根据分析的 command，在 dispatchers.go 中选择对应的函数进行处理 (dispatchers.go)。

(4) 处理完所有的命令，如果需要打标签，则给最后的镜像打上 tag，结束。

➤ Dockerfile 逆向

通过 docker history image 可以看到该镜像的历史来源。即使没有 Dockerfile，也可以通过 history 来逆向产生 Dockerfile。

执行 docker images 查看镜像并且得到镜像的 ID：

```
[root@pek-docker-01 ~]# docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
jhipster/jhipster	master	0bef6d7615d9	2 days ago	1.07 GB
ubuntu	14.04	3aa18c7568fc	11 days ago	188 MB
zookeeper	latest	f2249a75c5d0	2 months ago	143 MB
hello-world	latest	1815c82652c0	5 months ago	1.84 kB
vmware/harbor-log	v1.1.2	9c46a7b5e517	8 months ago	192 MB
jplock/zookeeper	3.4.8	80e2f8f23445	20 months ago	155 MB

然后执行 docker history 725dcfab7d63，就可以得到各个镜像的历史来源。

```
[root@pek-docker-01 ~]# docker history 725dcfab7d63
```

IMAGE	CREATED	CREATED BY	SIZE	COMMENT
725dcfab7d63	12 days ago	/bin/sh -c #(nop) CMD ["/hello"]		0 B
<missing>	12 days ago	/bin/sh -c #(nop) COPY file:b65349dad8105c...	1.84 kB	

4.4.4 构建 Java 环境

首先创建一个 Dockerfile 文件，定义作者并且注释清楚这个文件的意义：

```
#####
# Dockerfile to build java container images
# Based on Ubuntu
# File Author / Maintainer
MAINTAINER zhangfeng cloudskyme@163.com
#####
```

设置基础镜像，可以是 Ubuntu 也可以是 Centos，根据自己公司的情况设置：

```
# Set the base image to centos
FROM centos
```

创建目录并将已经下载好的 JDK 放到和 Dockerfile 相同的目录下，因为下载地址有可能会变动，尤其是 JDK。

将已经下载好的 JDK 上传到要构建 Dockerfile 的目录下，目录下应该包含 3 个文件：

```
[root@pek-docker-01 docker]# ls
chapter-04-07-0.0.1-SNAPSHOT.jar Dockerfile jdk-8u152-linux-x64.tar.gz
```

要构建的 Dockerfile、要打包的 JDK 和一个可运行的 jar 文件。

完整的 Dockerfile 内容如下：

```
MAINTAINER zhangfeng cloudskyme@163.com

FROM centos

RUN mkdir /var/tmp/jdk
COPY jdk-8u152-linux-x64.tar.gz /var/tmp/jdk
RUN tar xzf /var/tmp/jdk/jdk-8u152-linux-x64.tar.gz -C /var/tmp/jdk
RUN rm -rf /var/tmp/jdk/jdk-8u152-linux-x64.tar.gz
```



```
ADD chapter-04-07-0.0.1-SNAPSHOT.jar app.jar

ENV JAVA_HOME=/var/tmp/jdkjdk1.8.0_152
ENV PATH=$JAVA_HOME/bin:$PATH
ENV CLASSPATH=.:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar

EXPOSE 8080

ENTRYPOINT ["java","-jar","/app.jar"]
```

在目录下执行“docker build -t jdk8run.”，注意最后有设定是当前目录的“.”。

输出构建信息：

```
Sending build context to Docker daemon 204.8 MB
Step 1/12 : MAINTAINER zhangfeng cloudskyme@163.com
Please provide a source image with `from` prior to commit
[root@pek-docker-01 docker]# docker build -t jdk8run .
Sending build context to Docker daemon 204.8 MB
Step 1/12 : MAINTAINER zhangfeng cloudskyme@163.com
Please provide a source image with `from` prior to commit
[root@pek-docker-01 docker]# vi Dockerfile
[root@pek-docker-01 docker]# docker build -t jdk8run .
Sending build context to Docker daemon 204.8 MB
Step 1/11 : FROM centos
latest: Pulling from library/centos
d9aaf4d82f24: Already exists
Digest: sha256:4565fe2dd7f4770e825d4bd9c761a81b26e49cc9e3c9631c58cfc3188be9505a
Status: Downloaded newer image for centos:latest
---> d123f4e55e12
Step 2/11 : RUN mkdir /var/tmp/jdk
---> Running in 01af4f7fd310
---> aac07ee37217
Removing intermediate container 01af4f7fd310
Step 3/11 : COPY jdk-8u152-linux-x64.tar.gz /var/tmp/jdk
---> 2241a5fc43da
Removing intermediate container 91f15f2fb069
Step 4/11 : RUN tar xzf /var/tmp/jdk/jdk-8u152-linux-x64.tar.gz -C /var/tmp/jdk
---> Running in 5dlfde540d98
```

```
---> ce495024163d
Removing intermediate container 5d1fde540d98
Step 5/11 : RUN rm -rf /var/tmp/jdk/jdk-8u152-linux-x64.tar.gz
---> Running in 8694fa8f8024
---> 05c678ac8fce
Removing intermediate container 8694fa8f8024
Step 6/11 : ADD chapter-04-07-0.0.1-SNAPSHOT.jar app.jar
---> 7be12d2214d8
Removing intermediate container 5c33f2fe933e
Step 7/11 : ENV JAVA_HOME /var/tmp/jdkjdk1.8.0_152
---> Running in 6933c0eae655
---> 41202ad30a92
Removing intermediate container 6933c0eae655
Step 8/11 : ENV PATH $JAVA_HOME/bin:$PATH
---> Running in 47cefae4e20d
---> 4b7db45df583
Removing intermediate container 47cefae4e20d
Step 9/11 : ENV CLASSPATH .:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
---> Running in 958237944c0f
---> a11275ccf6cc
Removing intermediate container 958237944c0f
Step 10/11 : EXPOSE 8080
---> Running in 442eacc48057
---> 694c329d4d6b
Removing intermediate container 442eacc48057
Step 11/11 : ENTRYPOINT java -jar /app.jar
---> Running in c7b6e55fb742
---> ef45153e11a8
Removing intermediate container c7b6e55fb742
Successfully built ef45153e11a8
```

执行 `docker run -p 8080:8080 jdk8run`，然后在外部就可以通过“`http://IP`”加端口访问应用了。

4.4.5 Dockerfile 小结

Dockerfile 提供了一种可编程的制作镜像的方案，我们可以在各种复杂的环境中根据自己公

司的特点，加上片段应用所使用的脚本，定制属于自己的 Docker 镜像，从而做到各种环境互不入侵，又能够相互合作，为构建自己公司的微服务环境添砖加瓦。

4.5 Docker 网络

4.5.1 网络模式

当 Docker 进程启动时，会在主机上创建一个名为 `docker0` 的虚拟网桥，此主机上启动的 Docker 容器会连接到这个虚拟网桥上。虚拟网桥的工作方式和物理交换机类似，这样主机上的所有容器就通过交换机连在了一个二层网络中。

Docker 有以下四种网络模式：

- `host` 模式，`docker run` 时使用 `--net=host` 指定。
- `container` 模式，`docker run` 时使用 `--net=container:NAME_or_ID` 指定。
- `none` 模式，`docker run` 时使用 `--net=none` 指定。
- `bridge` 模式，`docker run` 时使用 `--net=bridge` 指定，默认设置。

4.5.2 link

`link` 是在两个 `contain` 之间建立一种父子关系，父 `container` 中的 `Web` 可以得到子 `container db` 上的信息。

通过 `link` 的方式创建容器，我们可以使用被 `link` 容器的别名进行访问，而不是通过 IP，解除了对 IP 的依赖。

不过，`link` 的方式只能解决单机容器间的互连，多机的情况下，需要通过别的方式进行连接。

在运行一个容器时，使用 `--link=container_name or id:name` 选项可以在此容器的 `/etc/hosts` 文件中增加一个额外的 `name` 主机名，这个名字为 `container_name` 的容器的 IP 地址的别名。这使得新容器的内部进程可以访问主机名为 `name` 的容器而不用知道它的 IP。

内网是走 `docker0` 的网桥，互相之间是 Ping 得通的，但是 `docker run` 建立容器时，它的 IP 地址是不可控制的，所以 Docker 用 `link` 的方式使 `Web` 能够访问到 `db` 中的数据。

4.5.3 跨主机访问

跨主机的容器访问目前市面上主流的解决方法有 flannel、weave、Pipework、Open vSwitch 等。下面就来分别认识一下这几种方案。

➤ Open vSwitch

Open vSwitch 是一个高质量的、多层虚拟交换机，使用开源 Apache 2.0 许可协议，由 Nicia Networks 开发，主要实现代码为可移植的 C 代码。它的目的是让大规模网络自动化可以通过编程扩展，同时仍然支持标准的管理接口和协议（例如，NetFlow、SFlow、SPAN、RSPAN、CLI、LAAP、802.lag）。

➤ Weave

Weave 是由 Zett.io 公司开发的，它能够创建一个虚拟网络，用于连接部署在多台主机上的 Docker 容器，这样容器就像被接入了同一个网络交换机一样，那些使用网络的应用程序不必去配置端口映射和链接等信息。外部设备能够访问 Weave 网络上的应用程序容器所提供的服务，同时已有的内部系统也能够暴露到应用程序容器上。Weave 能够穿透防火墙并运行在部分连接的网络上。另外，Weave 的通信支持加密，所以用户可以从一个不受信任的网络连接到主机。

➤ Flannel

Flannel 是 CoreOS 团队针对 Kubernetes 设计的一个网络规划服务，简单来说，它的功能是让集群中的不同节点主机创建的 Docker 容器都具有全集群唯一的虚拟 IP 地址。

4.6 Docker 数据卷

容器中管理数据主要有两种方式：

- 数据卷（Data Volumes）；
- 数据卷容器（Data Volumes Containers）。

4.6.1 数据卷

数据卷是一个特殊的目录，它将主机目录直接映射进容器，可供一个或多个容器使用，如图 4-5 所示。

数据卷设计的目的是为了数据的持久化，它完全独立与容器的生命周期。因此，容器删除时，不会删除其挂载的数据卷，也不会存在类似的垃圾机制对容器存在的数据卷进行处理。

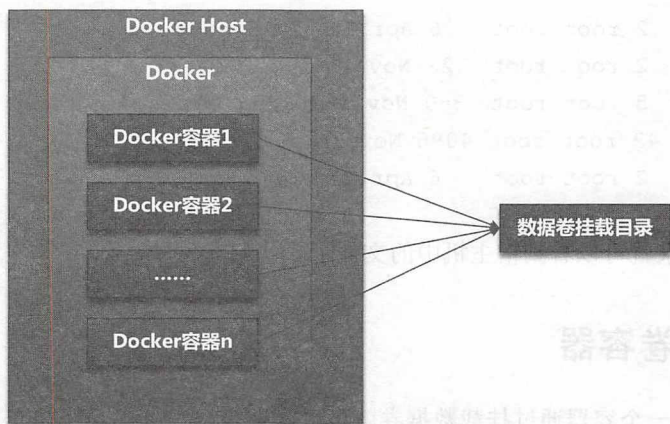


图 4-5 Docker 的数据卷

数据卷的特性:

- 数据卷在容器启动时初始化, 如果容器使用的镜像在挂载点包含了数据, 这些数据会复制到新初始化的数据卷中。
- 数据卷可以在容器之间共享和重用。
- 可以对数据卷里的内容直接修改, 修改会马上生效, 无论是容器内操作还是本地操作。
- 对数据卷的更新不会影响镜像的更新。
- 数据卷会一直存在, 即使挂载数据卷的容器已经被删除。

在 `docker run` 执行时添加 `-v` 参数, 冒号前为宿主机目录, 必须为绝对路径, 冒号后为镜像内挂载的路径。

在本地目录创建 `/root/data`, 然后将其挂载到容器中, 并且指定挂载的目录名。因为默认没有 `ls`, 所以需要 `apt-get` 安装 `ls`。

```
[root@pek-docker-01 data]# docker run -it -v /root/data:/data1 ubuntu /bin/bash
root@483926ff523e:/# apt-get install ls
Reading package lists... Done
Building dependency tree
Reading state information... Done
```

执行 `ls -l`, 就可以查看到挂载的目录了, 然后创建文件, 验证在 Docker 中是否能够看到。

```
root@4ba7b2096320:/# ls -l
total 12
drwxr-xr-x  2 root root 4096 Oct  6 01:38 bin
```

```
drwxr-xr-x  2 root root    6 Apr 12  2016 boot
drwxr-xr-x  2 root root   22 Nov 16 01:58 data1
drwxr-xr-x  5 root root  360 Nov 16 01:59 dev
drwxr-xr-x 42 root root 4096 Nov 16 01:59 etc
drwxr-xr-x  2 root root    6 Apr 12  2016 home
```

进入 data1 目录就可以看到宿主机中的文件了。

4.6.2 数据卷容器

一个目录或者一个容器通过挂载数据卷就可以实现容器与外部系统的交互了，但是如果多个容器想实现数据的共享又该怎么办呢？

Docker 提供了一种挂载数据卷的容器，叫作数据卷容器，其他容器能够通过挂载这个容器实现数据共享，如图 4-6 所示。

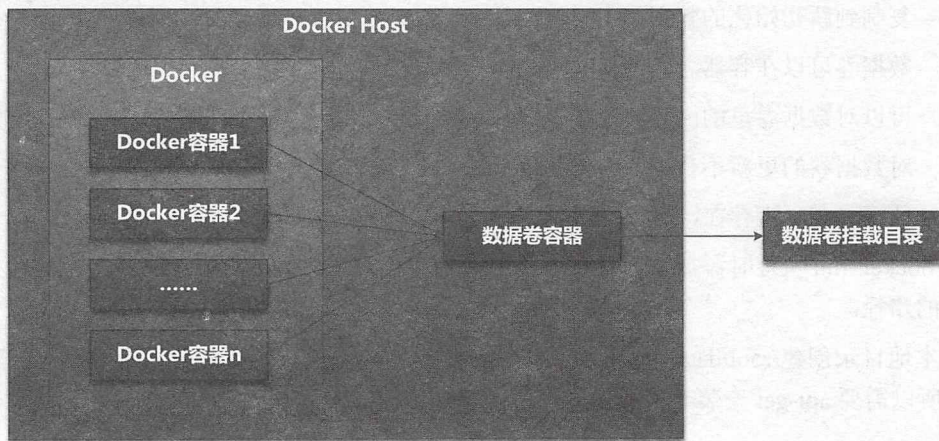


图 4-6 数据卷容器

可以利用数据卷容器对其中的数据卷进行备份、恢复，以实现数据的迁移。

4.7 Spring Boot 与 Docker

介绍

通过上面章节的介绍，我们已经了解了 Spring 和 Docker 的相关特性，那么这两个强大的开源框架是不是能够有机地整合在一起呢？答案是肯定的。

实现

(1) 新建 Spring Boot 工程 chapter-04-07，并且添加 Web 依赖。

(2) 为工程添加 Dockerfile-maven-plugin 插件，并且指定工程名称。添加插件代码如下，先添加镜像名称：

```
<properties>
  <docker.image.prefix>Springbootdocker</docker.image.prefix>
</properties>
```

然后添加 Dockerfile 构建插件：

```
<plugin>
  <groupId>com.spotify</groupId>
  <artifactId>dockerfile-maven-plugin</artifactId>
  <version>1.3.4</version>
  <configuration>
    <repository>${docker.image.prefix}/${project.artifactId}</repository>
  </configuration>
</plugin>
```

(3) 添加相关测试代码，访问指定的端口，可以直接返回“Hello Spring Boot And Docker World”。添加测试代码：

```
@RequestMapping("/")
public String home() {
    return "Hello Spring Boot And Docker World";
}
```

(4) 添加 Dockerfile 文件，Dockerfile 文件内容如下：

```
FROM java:8
MAINTAINER zhangfeng "cloudskyme@163.com"
VOLUME /tmp
ADD target/chapter-04-07-0.0.1-SNAPSHOT.jar app.jar
ENV JAVA_OPTS=""
ENTRYPOINT exec java $JAVA_OPTS -Djava.security.egd=file:/dev/./urandom
-jar /app.jar
```

测试

需要在一台安装有 Docker 环境的机器上进行测试，机器上同时安装有 Maven 和 Git 环境。
下载工程源代码，并且执行编译 `mvn clean install`，构建成功。

(1) 进入工程目录执行：

```
mvn package && java -jar target/chapter-04-07-0.0.1-SNAPSHOT.jar
```

构建成功后，可以看到输出日志：

```
2017-11-16 17:41:49.327 INFO 23318 --- [           main]
o.s.j.e.a.AnnotationMBeanExporter      : Registering beans for JMX exposure on
startup
2017-11-16 17:41:49.340 INFO 23318 --- [           main]
o.s.c.support.DefaultLifecycleProcessor : Starting beans in phase 0
2017-11-16 17:41:49.502 INFO 23318 --- [           main]
s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080
(http)
2017-11-16 17:41:49.510 INFO 23318 --- [           main]
com.cloud.skyme.Chapter0407Application : Started Chapter0407Application in
4.673 seconds (JVM running for 5.288)
```

打开浏览器输入主机的 IP 加端口，可以看到返回的结果：

```
Hello Spring Boot And Docker World
```

(2) 构建 Docker 镜像，使用命令：

```
mvn install dockerfile:build
```

可以看到如下输出：

```
[INFO] Building Docker context /root/microservice/microservice/chapter-04-07
[INFO]
[INFO] Image will be built as Springbootdocker/chapter-04-07:latest
[INFO]
[INFO] Step 1/6 : FROM java:8
[INFO] Pulling from library/java
[INFO] Digest:
```



```

sha256:c1ff613e8ba25833d2e1940da0940c3824f03f802c449f3d1815a66b7f8c0e9d
[INFO] Status: Image is up to date for java:8
[INFO] ---> d23bdf5b1b1b
[INFO] Step 2/6 : MAINTAINER zhangfeng "cloudskyme@163.com"
[INFO] ---> Using cache
[INFO] ---> 94a640dc0346
[INFO] Step 3/6 : VOLUME /tmp
[INFO] ---> Using cache
[INFO] ---> ec0093f6490
[INFO] Step 4/6 : ADD target/chapter-04-07-0.0.1-SNAPSHOT.jar app.jar
[INFO] ---> 70f5039a822c
[INFO] Removing intermediate container 4ff284fca5dd
[INFO] Step 5/6 : ENV JAVA_OPTS ""
[INFO] ---> Running in 4a42fa62fa29
[INFO] ---> 3b94cb44b02b
[INFO] Removing intermediate container 4a42fa62fa29
[INFO] Step 6/6 : ENTRYPOINT exec java $JAVA_OPTS -Djava.security.egd=
file:/dev/./urandom -jar /app.jar
[INFO] ---> Running in 74fb96402761
[INFO] ---> 72b287c3cb64
[INFO] Removing intermediate container 74fb96402761
[INFO] Successfully built 72b287c3cb64
[INFO]
[INFO] Detected build of image with id 72b287c3cb64
[INFO] Building jar: /root/microservice/microservice/chapter-04-07/
target/chapter-04-07-0.0.1-SNAPSHOT-docker-info.jar
[INFO] Successfully built Springbootdocker/chapter-04-07:latest
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 21.284 s
[INFO] Finished at: 2017-11-16T17:46:55+08:00
[INFO] Final Memory: 37M/450M
[INFO] -----

```

构建成功后，输入 `docker images`，查看镜像是否构建：

```
[root@pek-docker-01 chapter-04-07]# docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
Springbootdocker/chapter-04-07	latest	72b287c3cb64	About a minute ago	658 MB

能够查看到构建的镜像信息。然后运行镜像并将端口映射到外边:

```
[root@pek-docker-01 chapter-04-07]# docker run -p 8080:8080 -t
Springbootdocker/chapter-04-07
```

启动完成后, 通过 `docker ps` 查看启动的应用。

打开浏览器输入主机的 IP 加端口, 可以看到返回的结果:

```
Hello Spring Boot And Docker World
```

总结

Spring Boot 和 Docker 的整合能够大大提高打包的发布效率, 在后面的章节中我们将把构建的镜像上传到私有仓库中, 实现多人共享。

4.8 搭建自己的镜像仓库

Docker 容器应用的开发和运行离不开可靠的镜像管理, 虽然 Docker 官方也提供了公共的镜像仓库, 但是从安全和效率等方面考虑, 部署我们私有环境内的 Registry 也是非常必要的。

私有镜像仓库就是企业内部的镜像仓库, 一般为提高内部交互和开发加速所用。这里推荐使用的是 Harbor。

Harbor 是一个由 VMware 中国研发团队开发并开源的企业级 Docker 镜像仓库管理系统, 它对原有的 Docker Register 服务进行了扩展, 添加了更多的企业用户所需要的功能。比如管理 UI、基于角色的访问控制 (Role Based Access Control)、AD/LDAP 集成和审计日志 (Auditlogging) 等, 同时还原生支持中文。

4.8.1 安装和启动

项目地址为 <https://github.com/vmware/harbor>。

安装流程如下。

(1) 安装 docker-compose。

下载指定版本的 docker-compose

```
$ curl -L https://github.com/docker/compose/releases/download/1.13.0/
```



```
docker-compose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose
```

对二进制文件赋予可执行权限

```
$ sudo chmod +x /usr/local/bin/docker-compose
```

测试 docker-compose 是否安装成功

```
$ docker-compose --version
```

```
docker-compose version 1.13.0, build 1719ceb
```

(2) 下载 Harbor 安装文件。

从 GitHub Harbor 官网 [release](#) 页面下载指定版本的安装包。

在线安装包

```
$ wget https://github.com/vmware/harbor/releases/download/v1.1.2/harbor-online-installer-v1.1.2.tgz
```

```
$ tar xvf harbor-online-installer-v1.1.2.tgz
```

离线安装包

```
$ wget https://github.com/vmware/harbor/releases/download/v1.1.2/harbor-offline-installer-v1.1.2.tgz
```

```
$ tar xvf harbor-offline-installer-v1.1.2.tgz
```

(3) 配置 Harbor，解压缩之后，目录下会生成 harbor.conf 文件，该文件就是 Harbor 的配置文件。

```
## Configuration file of Harbor
```

```
# hostname 设置访问地址，可以使用 IP、域名，不可以设置为 127.0.0.1 或 localhost
hostname = 10.120.10.120
```

```
# 访问协议，默认是 HTTP，也可以设置 HTTPS，如果设置 HTTPS，则 Nginx SSL 需要设置为 on
ui_url_protocol = http
```

```
# MySQL 数据库 root 用户默认密码为 root123，实际使用时可以修改
db_password = root
```

```
max_job_workers = 3
```

```
customize_cert = on
```

```
ssl_cert = /data/cert/server.crt
```

```
ssl_cert_key = /data/cert/server.key
```

```
secretkey_path = /data
```

```
admiral_url = NA

# 邮件设置, 发送重置密码邮件时使用
email_identity =
email_server = smtp.mydomain.com
email_server_port = 25
email_username = sample_admin@mydomain.com
email_password = abc
email_from = admin <sample_admin@mydomain.com>
email_ssl = false

# 启动 Harbor 后, 管理员 UI 登录的密码默认是 Harbor12345
harbor_admin_password = Harbor12345

# 认证方式, 这里支持多种认证方式, 如 LDAP、本地存储、数据库认证。默认是 db_auth, mysql
数据库认证
auth_mode = db_auth

# LDAP 认证时配置项
#ldap_url = ldaps://ldap.mydomain.com
#ldap_searchdn = uid=searchuser,ou=people,dc=mydomain,dc=com
#ldap_search_pwd = password
#ldap_basedn = ou=people,dc=mydomain,dc=com
#ldap_filter = (objectClass=person)
#ldap_uid = uid
#ldap_scope = 3
#ldap_timeout = 5

# 是否开启自注册
self_registration = on

# Token 有效时间, 默认为 30 分钟
token_expiration = 30

# 用户创建项目权限控制, 默认是 everyone(所有人), 也可以设置为 adminonly(只能管理员)
project_creation_restriction = everyone

verify_remote_cert = on
```


(4) 启动 Harbor，修改完配置文件后，在当前目录执行“./install.sh, Harbor”，服务就会根据当期目录下的 docker-compose.yml 开始下载依赖的镜像，检测并按照顺序依次启动各个服务。Harbor 依赖的镜像及启动服务如下：

# docker images				
REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
vmware/harbor-jobservice	v1.1.2	ac332f9bd31c	10 days ago	162.9 MB
vmware/harbor-ui	v1.1.2	803897be484a	10 days ago	182.9 MB
vmware/harbor-adminserver	v1.1.2	360b214594e7	10 days ago	141.6 MB
vmware/harbor-db	v1.1.2	6f71ee20fe0c	10 days ago	328.5 MB
vmware/registry	2.6.1-photon	0f6c96580032	4 weeks ago	150.3 MB
vmware/harbor-notary-db	mariadb-10.1.10	64ed814665c6	10 weeks ago	324.1 MB
vmware/nginx	1.11.5-patched	8ddadb143133	10 weeks ago	199.2 MB
vmware/notary-photon	signer-0.5.0	b1eda7d10640	11 weeks ago	155.7 MB
vmware/notary-photon	server-0.5.0	6e2646682e3c	3 months ago	156.9 MB
vmware/harbor-log	v1.1.2	9c46a7b5e517	4 months ago	192.4 MB
photon	1.0	e6e4e4a2ba1b	11 months ago	127.5 MB

# docker-compose ps			
Name	Command	State	Ports

harbor-adminserver	/harbor/harbor_adminserver	Up	
harbor-db	docker-entrypoint.sh mysqld	Up	3306/tcp
harbor-jobservice	/harbor/harbor_jobservice	Up	
harbor-log	/bin/sh -c crond && rm -f ...	Up	127.0.0.1:1514->514/tcp
harbor-ui	/harbor/harbor_ui	Up	
nginx	nginx -g daemon off;	Up	0.0.0.0:443->443/tcp,
0.0.0.0:4443->4443/tcp,			0.0.0.0:80->80/tcp
registry	/entrypoint.sh serve /etc/ ...	Up	5000/tcp

启动完成后，可以发现 Harbor 由 6 个容器组成。

- harbor_ui: Harbor 的核心服务。
- harbor_log: 运行 rsyslog 的容器，进行日志收集。
- harbor_mysql: 由官方 MySQL 镜像构成的数据库容器。
- nginx: 使用 Nginx 做反向代理。

- registry: 官方的 Docker registry。
- harbor_jobervice: Harbor 的任务管理服务。

4.8.2 使用

默认情况下 Harbor 使用 HTTP 进行访问,如果是生产环境,建议使用 HTTPS,以确保安全。

安装完成后,打开浏览器,输入地址 `http://IP` 就可以得到 Harbor 的登录界面。

界面可以设置英文和中文,可以根据自己公司的情况进行相应的设置。

登录进入到控制台界面如图 4-7 所示。

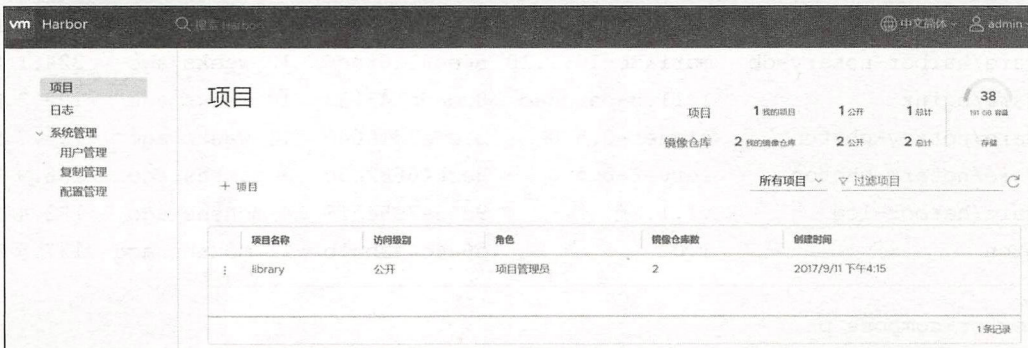


图 4-7 Harbor 控制台页面

系统各个模块如下。

- 项目: 新增/删除项目, 查看镜像仓库, 给项目添加成员、查看操作日志、复制项目等。
- 日志: 仓库各个镜像 create、push、pull 等操作日志。
- 系统管理:
 - 用户管理——新增/删除用户、设置管理员等。
 - 复制管理——新增/删除从库目标、新建/删除/启停复制规则等。
 - 配置管理——认证模式、复制、邮箱设置、系统设置等。
- 其他设置:
 - 用户设置——修改用户名、邮箱、名称信息。
 - 修改密码——修改用户密码。

注意: 非系统管理员用户登录, 只能看到有权限的项目和日志, 其他模块不可见。

如果需要上传镜像, 则需要使用 `docker login xx.xxx.xx.xx:5000`, 然后提示输入用户名和密码。

码，登录成功后，使用：

```
docker pull xx.xxx.xx.xx:5000/zhangfeng/node
```

向私服上传镜像文件。

4.9 Kubernetes

Swarm 为 Docker 自行开发的容器调度工具，2017 年成为 Docker 平台的内建工具。不过，有鉴于 Kubernetes 已成为最受欢迎的容器调度工具，再加上 Docker 用户也希望能够更方便地使用 Kubernetes，让 Docker 终于宣布支援 Kubernetes。

容器调度工具的竞争局面中，Kubernetes 可以说是站稳龙头，不只各家厂商抢着支援，连 Docker 都开始支援 Kubernetes，包含企业版 Docker、支持 Windows 与 Mac 的 Docker 社群版，以及 Moby 专案，用户可自行选择通过 Kubernetes 或 Swarm 来调度及管理容器任务。

那么 Kubernetes 到底是什么？为什么连 Docker 官方都宣布开始支持，我们一起来一探究竟。

Kubernetes (k8s) 是自动化容器操作的开源平台，这些操作包括部署、调度和节点集群间扩展。如果你曾经用过 Docker 容器技术部署容器，那么可以将 Docker 看作 Kubernetes 内部使用的低级别组件。Kubernetes 不仅仅支持 Docker，还支持 Rocket，这是另一种容器技术。

使用 Kubernetes 可以：

- 自动化容器的部署和复制；
- 随时扩展或收缩容器规模；
- 将容器组织成组，并且提供容器间的负载均衡；
- 很容易地升级应用程序容器的新版本；
- 提供容器弹性，如果容器失效就替换它，等等。

Kubernetes 集群组件如下。

- etcd：一个高可用的 K/V 键值对存储和服务发现系统。
- flannel：实现跨主机的容器网络的通信。
- kube-apiserver：提供 Kubernetes 集群的 API 调用。
- kube-controller-manager：确保集群服务。
- kube-scheduler：调度容器，分配到 Node。
- kubelet：在 Node 节点上按照配置文件中定义的容器规格启动容器。

- kube-proxy: 提供网络代理服务。

Kubernetes 的主要概念如下。

➤ Pods

Pod 是 Kubernetes 的基本操作单元, 把相关的一个或多个容器构成一个 Pod, 通常 Pod 里的容器运行相同的应用。Pod 包含的容器运行在同一个 Minion (Host) 上, 看作一个统一管理单元, 共享相同的 volumes、network namespace/IP 和 Port 空间。

➤ Services

Services 也是 Kubernetes 的基本操作单元, 是真实应用服务的抽象, 每一个服务后面都有很多对应的容器来支持, 通过 Proxy 的 port 和 selector 服务决定服务请求传递给后端提供服务的容器, 对外表现为一个单一访问接口, 外部不需要了解后端如何运行, 这给扩展或维护后端带来很大的好处。

➤ Replication Controllers

Replication Controller 确保任何时候 Kubernetes 集群中有指定数量的 Pod 副本 (replicas) 在运行, 如果少于指定数量的 Pod 副本 (replicas), Replication Controller 会启动新的 Container, 反之会“杀死”多余的以保证数量不变。Replication Controller 使用预先定义的 Pod 模板创建 pods, 一旦创建成功, Pod 模板和创建的 pods 没有任何关联, 可以修改 Pod 模板而不会对已创建 pods 有任何影响, 也可以直接更新通过 Replication Controller 创建的 pods。

➤ Labels

Labels 是用于区分 Pod、Service、Replication Controller 的 key/value 键值对, Pod、Service、Replication Controller 可以有多个 Label, 但是每个 Label 的 key 只能对应一个 value。Labels 是 Service 和 Replication Controller 运行的基础, 为了将访问 Service 的请求转发给后端提供服务的多个容器, 正是通过标识容器的 Labels 来选择正确的容器。同样, Replication Controller 也使用 Labels 来管理通过 Pod 模板创建的一组容器, 这样 Replication Controller 可以更加容易、方便地管理多个容器, 无论有多少容器。

Kubernetes 集群包括两种类型资源:

- Master 节点——协调控制整个集群。
- Nodes 节点——运行应用的工作节点。

Master 负责集群的管理, 协调集群中的所有行为/活动。例如, 应用的运行、修改、更新等。节点 (Node) 作为 Kubernetes 集群中的工作节点, 可以是 VM 虚拟机、物理机。每个 Node 上都有一个 Kubelet, 用于管理 Node 节点与 Kubernetes Master 通信。每个 Node 节点上至少还要运行 container runtime (比如 Docker 或者 rkt)。

在 Kubernetes 上部署应用程序时，会先通知 master 启动容器中的应用程序，master 调度容器在集群的节点上运行，Node 节点使用 master 公开的 Kubernetes API 与主节点进行通信。最终用户还可以直接使用 Kubernetes API 与集群进行交互。

在传统的概念当中，Docker 是简单易用的，Kubernetes 是复杂强大的，如果部署起来需要花费很长的时间，但是事实不是这样的。如果是前期测试或者尝鲜，可以使用单机搭建 Kubernetes 来了解整体的运行情况，这个工具就是 Minikube。

Minikube 的特点：

- Minikube 是一种方便在本地运行 Kubernetes 的工具。
- Minikube 是可以在 VM 中运行单节点的 Kubernetes 集群。
- Minikube 是为了开发或测试在本地启动一个节点的 kubernetes 集群。
- 可以工作在 Windows、Linux、MacOS 下。

项目地址：<https://github.com/kubernetes/minikube>。

首先准备一台 Linux 服务器，安装好 Docker。

安装 minikube，使用下面的命令：

```
curl -Lo minikube https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64 && chmod +x minikube && sudo mv minikube /usr/local/bin/
```

启动 minikube：

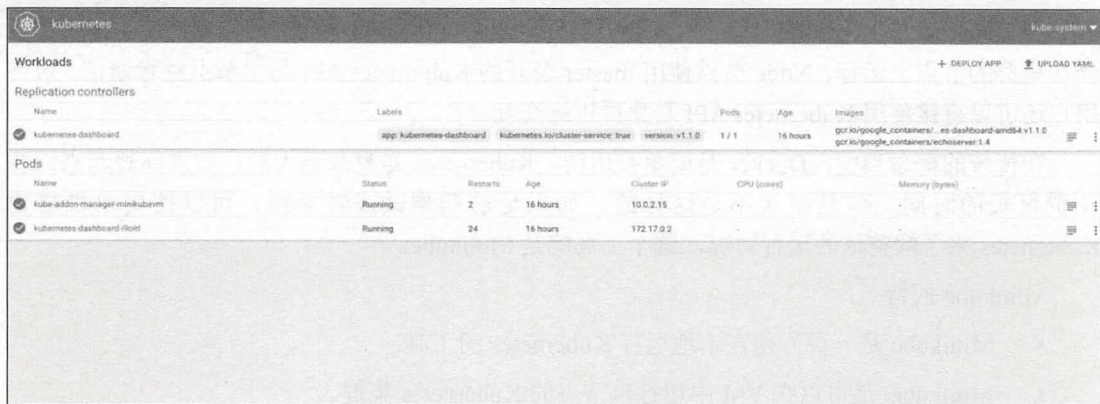
```
minikube start
```

打开 Kubernetes 控制台：

```
minikube dashboard
```

打开浏览器看到如图 4-8 所示的界面。

用户使用 Minikube CLI 管理虚拟机上的 Kubernetes 环境，比如启动、停止、删除、获取状态等。一旦 Minikube 虚拟机启动，用户就可以使用熟悉的 Kubectl CLI 在 Kubernetes 集群上执行操作。



The screenshot shows the Kubernetes dashboard interface. At the top, there's a header with the Kubernetes logo and 'kube-system'. Below it, the 'Workloads' section is active, showing a table of 'Replication controllers'. The table has columns for Name, Labels, Pods, Age, and Images. One entry is visible: 'kubernetes-dashboard' with 1/1 pods and an age of 16 hours. Below this, the 'Pods' section is active, showing a table with columns for Name, Status, Restarts, Age, Cluster IP, CPU (cores), and Memory (bytes). Two pods are listed: 'kubernetes-dashboard-minikubem' (Running, 2 restarts) and 'kubernetes-dashboard-ibot' (Running, 24 restarts).

Replication controllers					
Name	Labels	Pods	Age	Images	
kubernetes-dashboard	app: kubernetes-dashboard, kubernetes.io/cluster-service: true, version: v1.1.0	1 / 1	16 hours	gcr.io/google_containers/...-v1.1.0	

Pods						
Name	Status	Restarts	Age	Cluster IP	CPU (cores)	Memory (bytes)
kubernetes-dashboard-minikubem	Running	2	16 hours	10.0.2.15		
kubernetes-dashboard-ibot	Running	24	16 hours	172.17.0.2		

图 4-8 Kubernetes 控制台

4.10 私有云整体架构

不管是大公司还是小公司，都希望有一套自己的私有云平台，能够最大化利用资源，而基于 Docker，使得构建这样的私有云平台变得更加容易。

接下来看一下私有云平台的架构，如图 4-9 所示。

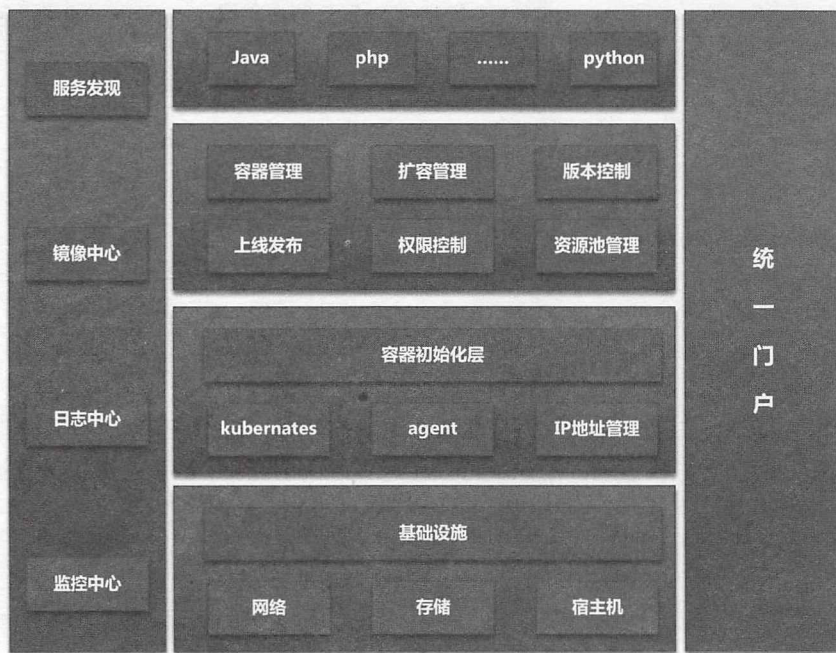


图 4-9 私有云平台架构

➤ 基础设施

顾名思义，就是指现有的服务器，以及由服务器构建的宿主机、存储和网络等资源。一般由硬件设备组成，处于所有应用的最底层。

➤ 容器层

基础设施之上提供了整个容器初始化层，容器初始化层包含 Kubernetes、Agent、IP 地址管理；上一节已经重点介绍了 Kubernetes。

Agent 部署在宿主机上，用于系统资源和底层基础设施的管理，包含监控采集、日志采集、容器限速等。IP 地址管理用于管理整个网络系统的 IP 资源。

➤ 资源管理

容器层之上是资源管理层，包含容器管理、扩容管理、版本控制、上线发布、权限控制和资源池管理等模块。能够根据资源的使用情况进行动态的分配，以满足更高并发的需求。

➤ 应用层

运行用户提交的业务实例，可以是任意编程语言。现在比较流行的是 Java、Python、.NET Core 和 PHP 等语言。平台能够为各种语言提供可运行的环境，并且支持一键部署。

➤ 基础组件

云平台为容器运行环境提供必备的基础组件，包含服务发现、镜像中心、日志中心、监控中心。服务发现用于将指定的请求路由到特定的服务上，镜像中心用于管理所有业务镜像，日志中心用于收集、分析业务中产生的日志数据，监控中心用于监控所有应用的运行状态，发现问题时及时告警处理。

➤ 统一门户

可视化的 UI 门户页面，规范化整个业务流程，简洁的用户流程，可动态管理整个云环境的所有资源。

4.11 小结

Docker 作为微服务中非常重要的一环，能够实现线上线下保持一致的环境，避免对特定的云供应商的依赖。能够有效地降低运维团队的负担，当访问量增大时，能够通过横向扩展集群的规模，做到弹性伸缩。

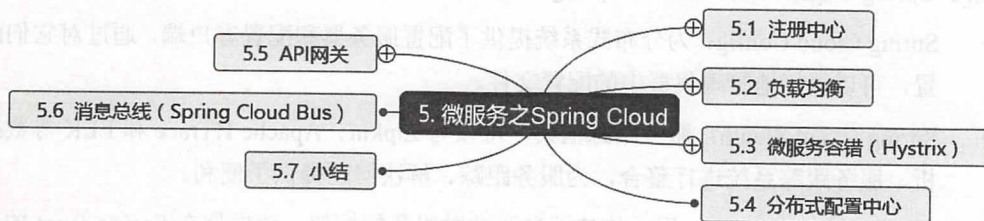
对于小型团队来说，借鉴别的团队成功的经验非常重要，使用 Docker，可以快速地应用其他团队的经验，也可以将自己团队的经验共享给其他团队。

Kubernetes 具备完善的集群管理能力，包括多层次的安全防护和准入机制，多租户应用支撑能力，透明的服务注册和服务发现机制，内建负载均衡器，故障发现和自我修复能力，服务滚动升级和在线扩容，可扩展的资源自动调度机制，多粒度的资源配额管理能力。

Kubernetes 还提供完善的管理工具，涵盖开发、部署测试、运维监控等各个环节。

5 chapter

第 5 章 微服务之 Spring Cloud



Spring Cloud 是一系列框架的有序集合，它利用 Spring Boot 的开发便利性巧妙地简化了分布式系统基础设施的开发，如服务发现注册、配置中心、消息总线、负载均衡、断路器、数据监控等，都可以用 Spring Boot 的开发风格做到一键启动和部署。

Spring Cloud 微服务工具包为开发者提供了分布式系统中的配置管理、服务发现、断路器、智能路由、微代理、控制总线等开发工具包。它的各个项目基于 Spring Boot，将 Netflix 的多个框架进行封装，并且通过自动配置的方式将这些框架绑定到 Spring 的环境中，从而简化了这些框架的使用。由于 Spring Boot 的简便性，使得我们在使用 Spring Cloud 时，很容易将 Netflix 各个框架整合进我们的项目中。Spring Cloud 下的“Spring Cloud Netflix”模块主要封装了 Netflix 的以下项目。

- Eureka: 基于 REST 服务的分布式中间件，主要用于服务管理。
- Hystrix: 容错框架，通过添加延迟阈值及容错的逻辑来帮助我们控制分布式系统间组件的交互。
- Feign: 一个 REST 客户端，目的是简化 Web Service 客户端的开发。
- Ribbon: 负载均衡框架，在微服务集群中为各个客户端的通信提供支持，它主要实现中间层应用程序的负载均衡。
- Zuul: 为微服务集群提供过代理、过滤、路由等功能。

除了 Spring Cloud Netflix 模块，Spring Cloud 还包括以下几个重要的模块。

- Spring Cloud Config: 为分布式系统提供了配置服务器和配置客户端，通过对它们的配置，可以很好地管理集群中的配置文件。
- Spring Cloud Sleuth: 服务跟踪框架，可以与 Zipkin、Apache HTrace 和 ELK 等数据分析、服务跟踪系统进行整合，为服务跟踪、解决问题提供了便利。
- Spring Cloud Stream: 用于构建消息驱动微服务的框架，该框架在 Spring Boot 的基础上，整合了“Spring Integration”来连接消息代理中间件。
- Spring Cloud Bus: 连接 RabbitMQ、Kafka 等消息代理的集群消息总线。

Spring Cloud 抛弃了 Dubbo 的 RPC 通信，采用的是基于 HTTP 的 REST 方式，这两种方式各有优劣。虽然牺牲了服务调用的性能，但也避免了上面提到的原生 RPC 带来的问题。而且 REST 相比 RPC 更为灵活，服务提供方和调用方的依赖只依靠一纸契约，不存在代码级别的强依赖，这在强调快速演化的微服务环境下显得更加合适。

Spring Cloud 的组件架构如图 5-1 所示。

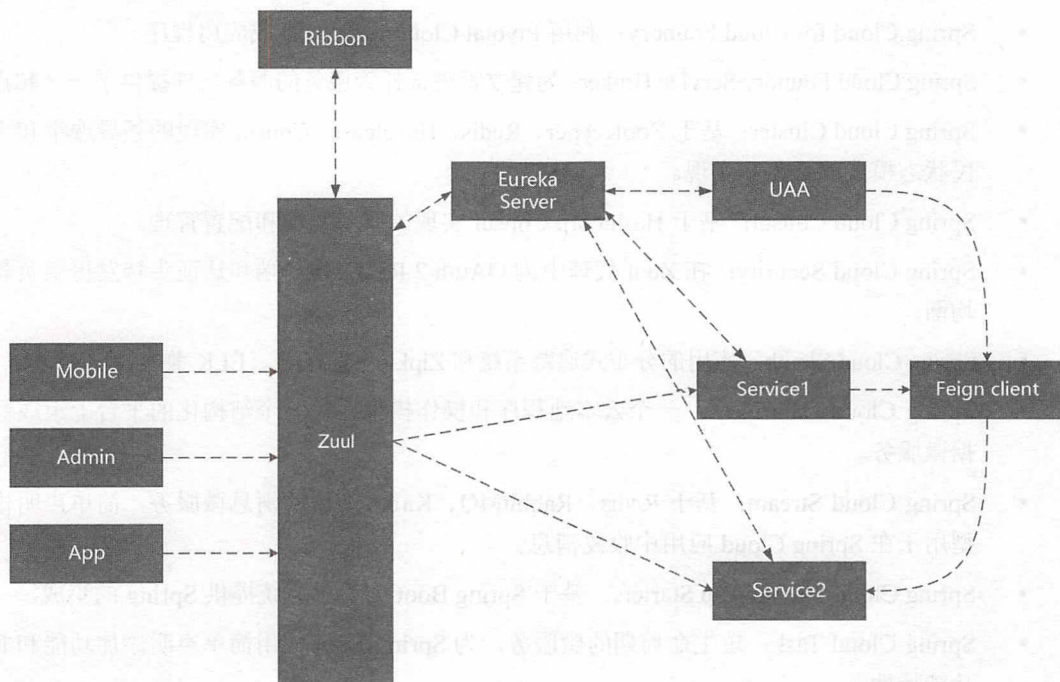


图 5-1 Spring Cloud 组件架构

各组件的运行流程如下：

- 所有请求都统一通过 API 网关（Zuu1）来访问内部服务。
- 网关接收到请求后，从注册中心（Eureka）获取可用服务。
- 由 Ribbon 进行均衡负载后，分发到后端的具体实例。
- 微服务之间通过 Feign 进行通信处理业务。
- Hystrix 负责处理服务超时熔断。
- Turbine 监控服务间的调用和熔断相关指标。

上面只是 Spring Cloud 体系的一部分，Spring Cloud 共集成了 19 个子项目，里面都包含一个或者多个第三方的组件或框架。

Spring Cloud 工具框架如下。

- Spring Cloud Config：配置中心，利用 Git 集中管理程序的配置。
- Spring Cloud Netflix：集成众多 Netflix 的开源软件。
- Spring Cloud Bus：消息总线，利用分布式消息将服务和 service 实例连接在一起，用于在一个集群中传播状态的变化。

- Spring Cloud for Cloud Foundry: 利用 Pivotal Cloudfoundry 集成应用程序。
- Spring Cloud Foundry Service Broker: 为建立管理云托管服务的服务代理提供了一个起点。
- Spring Cloud Cluster: 基于 ZooKeeper、Redis、Hazelcast、Consul 实现的领导选举和平民状态模式的抽象和实现。
- Spring Cloud Consul: 基于 Hashicorp Consul 实现的服务发现和配置管理。
- Spring Cloud Security: 在 Zuul 代理中为 OAuth 2 REST 客户端和认证头转发提供负载均衡。
- Spring Cloud Sleuth: 应用的分布式追踪系统和 Zipkin、HTrace、ELK 兼容。
- Spring Cloud Data Flow: 一个云本地程序和操作模型，在一个结构化的平台上组成数据微服务。
- Spring Cloud Stream: 基于 Redis、RabbitMQ、Kafka 实现的消息微服务，简单声明模型用于在 Spring Cloud 应用中收发消息。
- Spring Cloud Stream App Starters: 基于 Spring Boot 为外部系统提供 Spring 的集成。
- Spring Cloud Task: 短生命周期的微服务，为 Spring Boot 应用简单声明添加功能和非功能特性。
- Spring Cloud Task: 任务调度框架。
- Spring Cloud ZooKeeper: 服务发现和配置管理基于 Apache ZooKeeper。
- Spring Cloud for Amazon Web Services: 快速和亚马逊网络服务集成。
- Spring Cloud Connectors: 便于 PaaS 应用在各种平台上连接到后端数据库和消息经纪服务。
- Spring Cloud Starters: 项目已经终止并且在 Angel.SR 2 后的版本和其他项目合并。
- Spring Cloud CLI: 基于 Spring Cloud CLI，可以以命令行方式快速建立云组件。

这些工具集还在不断地增加。

5.1 注册中心

CAP 理论指出，一个分布式系统不可能同时满足 C（一致性）、A（可用性）和 P（分区容错性）。由于分区容错性在分布式系统中是必须要保证的，因此我们只能在 A 和 C 之间进行权衡。

那么如何选择一款适合自己的注册中心呢？就让我们来看一下常用的注册中心。注册中心就像是书的目录，而章节的内容就是具体的实现，当服务之间互相调用时，相当于先通过注册中心找到对应的目录，然后去调用相应的实现完成功能。

5.1.1 常用的注册中心

常用的注册中心包括 ZooKeeper、Eureka、etcd 和 Consul。

➤ ZooKeeper

ZooKeeper 是一种为分布式应用所设计的高可用、高性能且一致的开源协调服务，它提供了一项基本服务：分布式锁服务。由于 ZooKeeper 的开源特性，后来开发者在分布式锁的基础上，摸索出了其他的使用方法：配置维护、组服务、分布式消息队列、分布式通知/协调等。ZooKeeper 性能上的特点决定了它能够用在大型的、分布式的系统当中。从可靠性方面来说，它并不会因为一个节点的错误而崩溃。除此之外，它严格的序列访问控制意味着复杂的控制原语可以应用在客户端上。

很多场景下 ZooKeeper 也作为 Service 发现服务解决方案。ZooKeeper 保证的是 CP，即任何时刻对 ZooKeeper 的访问请求能得到一致的数据结果，同时系统对网络分割具备容错性，但是它不能保证每次服务请求的可用性。

➤ Eureka

Eureka 是 Netflix 开发的服务发现框架，Spring Cloud 将它集成在自己的子项目 Spring-cloud-netflix 中，实现 Spring Cloud 的服务发现功能。Eureka Server 会提供服务注册功能，各个服务节点启动后，会在 Eureka Server 中进行注册，这样 Eureka Server 中就有了所有服务节点的信息，并且 Eureka 有监控页面，可以在页面中直观地看到所有注册的服务的情况。同时 Eureka 有心跳机制，当某个节点服务在规定时间内没有发送心跳信号时，Eureka 会从服务注册表中把这个服务节点移除。Eureka 还提供了客户端缓存的机制，即使所有的 Eureka Server 都挂掉，客户端仍然可以利用缓存中的信息调用服务节点的服务。Eureka 一般配合 Ribbon 进行使用，Ribbon 提供了客户端负载均衡的功能，Ribbon 利用从 Eureka 中读取到的服务信息，在调用服务节点提供的服务时，会合理地进行负载。Eureka 遵守的就是 AP 原则。

➤ etcd

etcd 是一个高可用的键值存储系统，主要用于共享配置和服务发现。etcd 是由 CoreOS 开发并维护的，灵感来自 ZooKeeper 和 Doozer，它使用 Go 语言编写，并通过 Raft 一致性算法处理日志复制以保证强一致性。Raft 是一个新的一致性算法，适用于分布式系统的日志复制，Raft 通过选举的方式来实现一致性。Google 的容器集群管理系统 Kubernetes、开源 PaaS 平台 Cloud Foundry 和 CoreOS 的 Fleet 都广泛使用了 etcd。在分布式系统中，如何管理节点间的状态一直是一个难题，etcd 像是专门为集群环境的服务发现和注册而设计的，它提供了数据 TTL 失效、数据改变监视、多值、目录监听、分布式锁原子操作等功能，可以方便地跟踪并管理集群节点的状态。

➤ Consul

Consul (<https://www.consul.io/downloads.html>) 是 HashiCorp 公司推出的开源工具, 用于实现分布式系统的服务发现与配置共享。对比其他分布式服务注册与发现的方案, Consul 的方案更“一站式”, 内置了服务注册与发现框架、分布一致性协议实现(Raft 算法)、健康检查、Key/Value 存储、多数据中心方案, 不再需要依赖其他工具(比如 ZooKeeper 等)。Consul 用 Golang 实现, 因此具有天然可移植性(支持 Linux、Windows 和 Mac OS X); 安装包仅包含一个可执行文件, 方便部署, 与 Docker 等轻量级容器可无缝配合。

在注册中心的选择方面, 可以针对自身业务的特点, 选择一款合适的注册中心, 如果团队原有框架基于 Dubbo, 那么 ZooKeeper 会更合适一些; 如果团队整体使用 Spring Cloud, 那么 Eureka 的优势就会更大一些; 如果团队对于容器化的要求比较高, 那么 etcd 和 Consul 都是不错的选择。由于我们的整体构建所需, 我们选择的是 Spring Cloud 中的 Eureka。

5.1.2 Eureka 介绍

Eureka 的一些概念如下。

➤ Register: 服务注册

当 Eureka 客户端向 Eureka Server 注册时, 它提供自身的元数据, 比如 IP 地址、端口、运行状况指示符 URL、主页等。

➤ Renew: 服务续约

Eureka 客户会每隔 30 秒发送一次心跳来续约。通过续约来告知 Eureka Server 该 Eureka 客户仍然存在, 没有出现问题。正常情况下, 如果 Eureka Server 在 90 秒后没有收到 Eureka 客户的续约, 则它会将实例从其注册表中删除。建议不要更改续约间隔。

➤ Fetch Registries: 获取注册列表信息

Eureka 客户端从服务器获取注册表信息, 并将其缓存在本地。客户端会使用该信息查找其他服务, 从而进行远程调用。该注册列表信息定期(每 30 秒)更新一次。每次返回的注册列表信息可能与 Eureka 客户端的缓存信息不同, Eureka 客户端会自动处理。如果由于某种原因导致注册列表信息不能及时匹配, 则 Eureka 客户端会重新获取整个注册表信息。Eureka 服务器缓存注册列表信息, 整个注册表及每个应用程序的信息都进行了压缩, 压缩内容和没有压缩的内容完全相同。Eureka 客户端和 Eureka 服务器可以使用 JSON / XML 格式进行通信。在默认的情况下, Eureka 客户端使用压缩 JSON 格式来获取注册列表的信息。

➤ Cancel: 服务下线

Eureka 客户端在程序关闭时向 Eureka 服务器发送取消请求。发送请求后, 该客户端实例信

息将从服务器的实例注册表中删除。该下线请求不会自动完成，它需要调用以下内容：

```
DiscoveryManager.getInstance().shutdownComponent();
```

➤ Eviction：服务剔除

在默认的情况下，当 Eureka 客户端连续 90 秒没有向 Eureka 服务器发送服务续约（即心跳）时，Eureka 服务器会将该服务实例从服务注册列表删除，即服务剔除。

Eureka 由多个 instance（服务实例）组成，这些服务实例可以分为两种：Eureka Server 和 Eureka Client。

Eureka Client 再分为 Service Provider 和 Service Consumer。

Eureka Server：服务的注册中心，负责维护注册的服务列表。

- Service Provider：服务提供方，作为一个 Eureka Client，向 Eureka Server 进行服务注册、续约和下线等操作，注册的主要数据包括服务名、机器 IP、端口号、域名等。
- Service Consumer：服务消费方，作为一个 Eureka Client，向 Eureka Server 获取 Service Provider 的注册信息，并通过远程调用与 Service Provider 进行通信。

Service Provider 和 Service Consumer 不是严格的概念，Service Consumer 也可以随时向 Eureka Server 注册，来让自己变成一个 Service Provider。

Eureka 程序构成如下。

- （1）纯正的 Servlet 应用，需构建成 war 包部署。
- （2）使用了 Jersey 框架实现自身的 RESTful HTTP 接口。
- （3）peer 之间的同步与服务的注册全部通过 HTTP 协议实现。
- （4）定时任务（发送心跳、定时清理过期服务、节点同步等）通过 JDK 自带的 Timer 实现。
- （5）内存缓存使用 Google 的 guava 包实现。

5.1.3 服务发现

在现在的软件开发中，如果对性能要求不是非常高，则一般使用 REST API 来开放服务的接口。

服务发现有如下两种模式。

客户端服务发现（Client-Side Discovery）如图 5-2 所示。

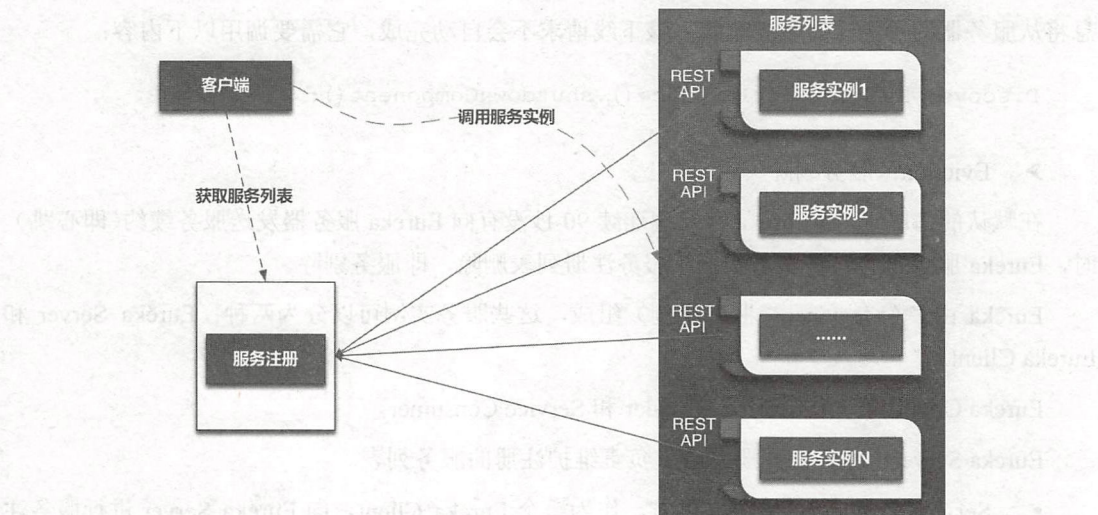


图 5-2 客户端服务发现

服务实例启动时会向服务注册中心进行注册，服务的注册中心能够看到所有注册的实例。客户端需要调用服务时，先到注册中心摘取可用的服务列表的地址，然后根据负载均衡算法，去获取一个可用的实例的地址来响应这次请求。

一个服务实例被启动，它的网络地址会被写到注册中心，当服务实例终止，会从注册表中删除。这个服务实例的注册表通过心跳机制动态刷新。

服务端服务发现（Server-Side Discovery）如图 5-3 所示。

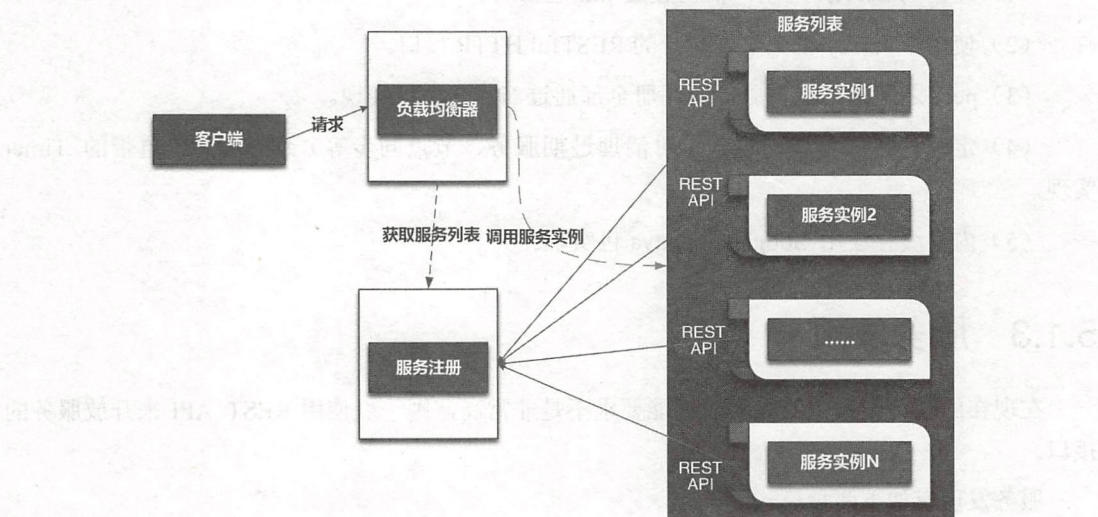


图 5-3 服务端服务发现

客户端通过负载均衡器向某个服务提出请求，负载均衡器向注册中心发出请求，将每个请求转发至可用的服务实例。和客户端发现一样，服务实例启动时在注册中心注册，当服务实例销毁时，会从服务注册表中删除。

使用服务器端服务发现，客户端无须关注发现的细节，只需要简单地向负载均衡器发送请求即可，实际上减少了编程语言框架需要完成的服务发现逻辑。缺点是除非部署环境能够提供负载均衡，否则负载均衡器是另外一个需要配置管理的高可用系统功能。目前比较流行的方式是使用 Nginx 来进行服务器端的负载均衡。

5.1.4 简单注册

介绍

Spring Cloud 针对服务注册与发现进行了一层抽象，并提供了三种实现：Eureka、Consul 和 ZooKeeper。目前支持得最好的就是 Eureka，其次是 Consul，最后是 ZooKeeper。

Eureka Server 可以运行多个实例来构建集群，解决单点问题，但不同于 ZooKeeper 的选举 leader 的过程，Eureka Server 采用的是 Peer to Peer 对等通信。这是一种去中心化的架构，无 master/slave 区分，每一个 Peer 都是对等的。在这种架构中，节点通过彼此互相注册来提高可用性，每个节点需要添加一个或多个有效的 serviceUrl 来指向其他节点。每个节点都可被视为其他节点的副本。

如果某台 Eureka Server 宕机，Eureka Client 的请求会自动切换到新的 Eureka Server 节点，当宕机的服务器重新恢复后，Eureka 会再次将其纳入服务器集群管理之中。当节点开始接收客户端请求时，所有的操作都会进行 replicateToPeer（节点间复制）操作，将请求复制到其他 Eureka Server 当前所知的所有节点中。

一个新的 Eureka Server 节点启动后，会首先尝试从邻近节点获取所有实例注册表信息，完成初始化。Eureka Server 通过 getEurekaServiceUrls() 方法获取所有的节点，并且会通过心跳续约的方式定期更新。默认配置下，如果 Eureka Server 在一定时间内没有接收到某个服务实例的心跳，则 Eureka Server 会注销该实例（默认为 90 秒，通过 eureka.instance.lease-expiration-duration-in-seconds 配置）。当 Eureka Server 节点在短时间内丢失过多的心跳时（比如发生了网络分区故障），那么这个节点就会进入自我保护模式。

Eureka 的高级架构如图 5-4 所示。

从图 5-4 可以看出在这个体系中有 2 个角色，即 Eureka Server 和 Eureka Client。而 Eureka Client 又分为 Application Service 和 Application Client，即服务提供者和服务消费者。每个区域有一个 Eureka 集群，并且每个区域至少有一个 Eureka 服务器可以处理区域故障，以防止服务器瘫痪。

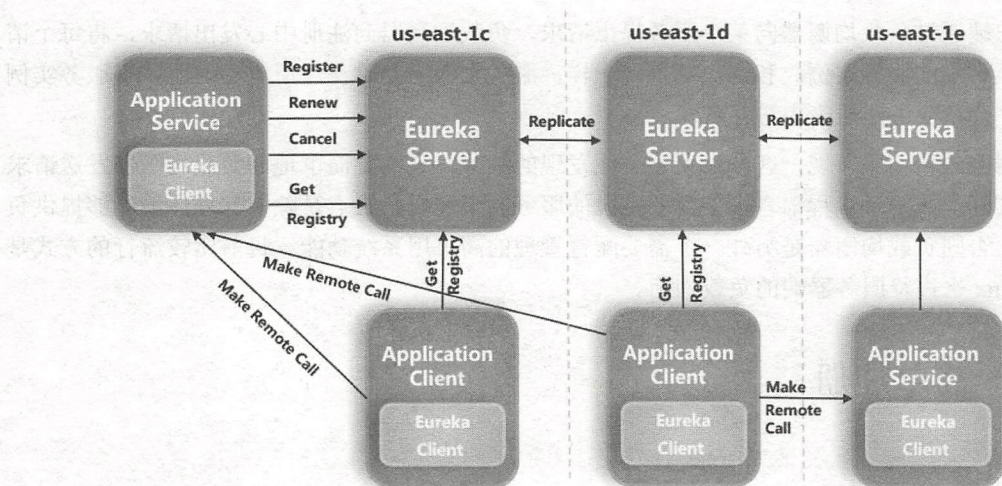


图 5-4 Eureka 高级架构

Eureka Client 向 Eureka Server 注册服务,并将自己的一些客户端信息发送给 Eureka Server。然后, Eureka Client 通过向 Eureka Server 发送心跳(每 30 秒)来续约服务。如果客户端持续不能续约,那么它将在大约 90 秒内从服务器注册表中删除。注册信息和续订被复制到集群中的 Eureka Server 所有节点。来自任何区域的 Eureka Client 都可以查找注册表信息(每 30 秒发生一次)。根据这些注册表信息, Application Client 可以远程调用 Application Service 来消费服务。

实现

- (1) 新建工程 chapter-05-01-eureka-server，并且添加 Eureka 依赖。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>Spring-cloud-starter-eureka-server</artifactId>
</dependency>
```

- (2) 在应用启动主类上添加@EnableEurekaServer, 然后在配置文件中添加如下配置。

```
server:
  port: 8761    #指定服务端口
eureka:
  client:
    registerWithEureka: false #是否将 Eureka 自身作为应用注册到 Eureka 注册中心
    fetchRegistry: false     #为 true 时，可以启动，但报异常: Cannot execute
request on any known server
```


这样就可以实现服务的注册中心了。

(3) 注册服务到注册中心，新建工程，并且添加依赖。

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>Spring-cloud-starter-eureka</artifactId>
</dependency>
```

(4) 在启动主类上添加@EnableDiscoveryClient，通过该注解，实现服务发现、注册。也可以使用@EnableEurekaClient，这两个的区别就是一个是通用的服务发现，可以发现 Eureka、Consul、ZooKeeper 中注册的服务，一个是专门为 Eureka 使用的。

(5) 添加示例，新建类 ServiceInstanceRestController，然后加入如下代码。

```
@RestController
public class ServiceInstanceRestController {
    @Autowired
    private DiscoveryClient discoveryClient;
    @RequestMapping("/service-instances/{applicationName}")
    public List<ServiceInstance> serviceInstancesByApplicationName
(@PathVariable String applicationName) {
        return this.discoveryClient.getInstances(applicationName);
    }
    @RequestMapping("/")
    public String sayhello() {
        return "hello eureka";
    }
}
```

(6) 添加配置文件，bootstrap.yml 和 application.yml。bootstrap.yml 先于 application.yml 加载，一般不变的东西配置在 bootstrap.yml 里，使用的方式是一样的。

bootstrap.yml 的配置文件内容如下：

```
bootstrap.yml
server:
  port: 8080
Spring:
```

```
application:
  name: cloud-client    #为应用起个名字, 该名字将注册到 Eureka 注册中心
```

Spring.application.name 很重要, 在以后的服务与服务之间一般都是根据这个 name 相互调用的。

application.yml 的配置文件内容如下:

```
eureka:
  client:
    serviceUrl:
      defaultZone: localhost:8761。 #注册中心的地址
```

多个 Eureka 服务地址可以用 “,” 号隔开。

测试

(1) 测试注册中心, 启动应用, 在启动完成后看到输出如下信息。

```
2017-11-20 17:36:15.383 INFO 574292 --- [ Thread-11]
o.s.c.n.e.server.EurekaServerBootstrap : Initialized server context
2017-11-20 17:36:15.390 INFO 574292 --- [ Thread-11]
e.s.EurekaServerInitializerConfiguration : Started Eureka Server
2017-11-20 17:36:15.632 INFO 574292 --- [ main]
s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8761
(http)
2017-11-20 17:36:15.633 INFO 574292 --- [ main]
.s.c.n.e.s.EurekaAutoServiceRegistration: Updating port to 8761
2017-11-20 17:36:15.642 INFO 574292 --- [ main]
c.c.s.Chapter0501EurekaserverApplication : Started
Chapter0501EurekaserverApplication in 20.156 seconds (JVM running for 24.261)
```

然后打开浏览器输入 <http://localhost:8761/>, 可以看到注册中心界面。

(2) 启动工程 chapter-05-01-eurekaregistry。

```
2017-11-21 10:22:10.360 INFO 11516 --- [nfoReplicator-0] com.netflix.
discovery.DiscoveryClient : DiscoveryClient_CLOUD-CLIENT/DESKTOP-E3I9LR5:
cloud-client:8080: registering service...
2017-11-21 10:22:10.579 INFO 11516 --- [ main]
s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
```


看到本地 8080 端口正常启动，然后访问 <http://localhost:8080/service-instances/cloud-client/>，可以看到元数据信息。

再次访问注册中心，能够看到应用上已经多了 CLOUD-CLIENT 的应用，说明此工程的服务已经注册到注册中心。

(3) 新建测试类 `Chapter0501EurekaserverApplicationTests`，并且添加如下代码。

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class Chapter0501EurekaserverApplicationTests {

    @LocalServerPort
    private int port;

    @Autowired
    private TestRestTemplate testRestTemplate;

    @Test
    public void shouldStartEurekaServer() {
        ResponseEntity<String> entity = this.testRestTemplate
            .getForEntity("http://localhost:" + this.port + "/eureka/apps",
                String.class);

        then(entity.getStatusCode()).isEqualTo(HttpStatus.OK);
    }
}
```

`RestTemplate` 简化了与 HTTP 服务器之间的通信，提供了一系列调用 Spring MVC REST 的接口，可以调用 HTTP 请求的 `WebService`，将结果转换成相应的对象类型。

首先输入 <http://localhost:8761/eureka/apps> 查看注册中心的元数据信息，然后通过单元测试访问测试与期望相符。运行单元测试，可以看到执行结果与期望相符，测试成功。

总结

注册中心可以做成高可用的，这取决于业务的规模，如果单纯用作注册中心，则 `Eureka` 比 `ZooKeeper` 有更多的优势，并且与 `Spring Cloud` 体系集成得更好。

5.2 负载均衡

负载均衡是云计算的基础组件，是网络流量的入口，其重要性不言而喻。

什么是负载均衡呢？用户输入的流量通过负载均衡器按照某种负载均衡算法把流量均匀地分散到后端的多个服务器上，接收到请求的服务器可以独立地响应请求，达到负载分担的目的。从应用场景上来说，常见的负载均衡模型有全局负载均衡和集群内负载均衡。

➤ 服务端负载均衡

负载均衡是处理高并发、缓解网络压力和进行服务端扩容的重要手段之一，但是一般情况下我们所说的负载均衡通常都是指服务端负载均衡，服务端负载均衡又分为两种，一种是硬件负载均衡，另一种是软件负载均衡。

硬件负载均衡主要通过服务器节点之间安装专门用于负载均衡的设备实现，常见的设备如 F5。

软件负载均衡则主要通过服务器上安装一些具有负载均衡功能的软件来完成请求分发进而实现负载均衡，常见的就是 Nginx。

无论是硬件负载均衡还是软件负载均衡，它的工作原理均如图 5-5 所示。

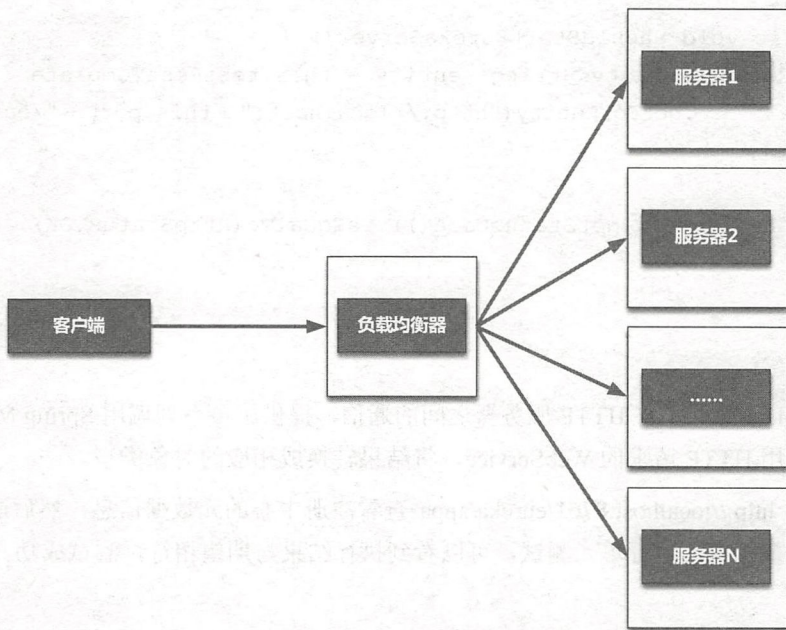


图 5-5 服务端负载均衡

无论是硬件负载均衡还是软件负载均衡，都会维护一个可用的服务端清单，然后通过心跳

机制来删除故障的服务端节点以保证清单中都是可以正常访问的服务端节点。当客户端的请求到达负载均衡服务器时，负载均衡服务器按照某种配置好的规则从可用服务端清单中选出一台服务器去处理客户端的请求，这就是服务端负载均衡。

➤ 客户端负载均衡

“Ribbon 是一个基于 HTTP 和 TCP 的客户端负载均衡器，当我们将 Ribbon 和 Eureka 一起使用时，Ribbon 会从 Eureka 注册中心去获取服务端列表，然后进行轮询访问以达到负载均衡的作用，客户端负载均衡中也需要心跳机制去维护服务端清单的有效性，当然这个过程需要配合服务注册中心一起完成。”

客户端负载均衡和服务端负载均衡最大的区别在于服务清单所存储的位置。在客户端负载均衡中，所有的客户端节点都有一份自己要访问的服务端清单，这些清单统统都是从 Eureka 服务注册中心获取的。在 Spring Cloud 中如果想要使用客户端负载均衡，方法很简单，开启 `@LoadBalanced` 注解即可，这样客户端在发起请求时会先自行选择一个服务端，向该服务端发起请求，从而实现负载均衡，如图 5-6 所示。

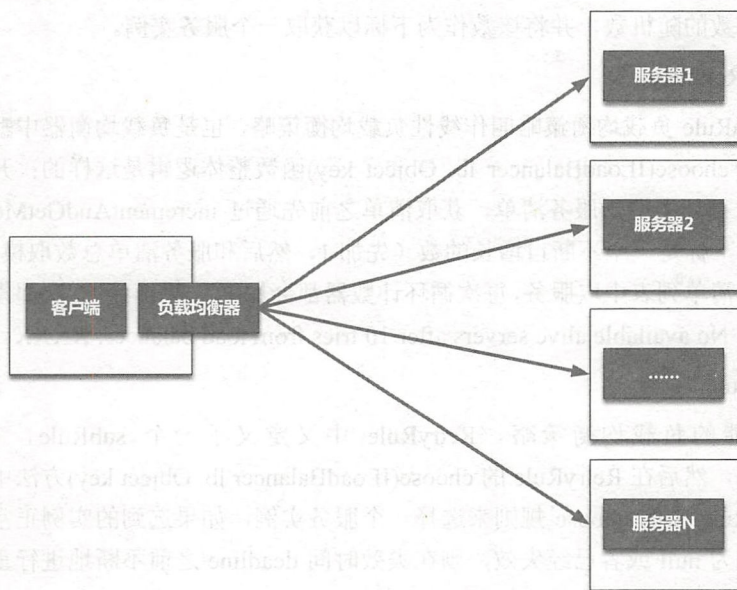


图 5-6 客户端负载均衡

这里重点要说的是软件负载均衡，即在不扩展硬件的情况下，通过软件的横向扩展达到承压的目的。

负载均衡策略：

- 简单轮询负载均衡；

- 加权响应时间负载均衡;
- 区域感知轮询负载均衡;
- 随机负载均衡。

5.2.1 Spring Cloud 的负载均衡实现

Spring Cloud 是如何实现负载均衡的呢?

Spring Cloud 的 `AbstractLoadBalancerRule` 是一个抽象类,里边主要定义了一个 `ILoadBalancer`,接下来看一下它的几个实现。

➤ RandomRule

这种负载均衡策略就是随机选择一个服务实例,在 `RandomRule` 的无参构造方法中初始化了一个 `Random` 对象,然后在它重写的 `choose` 方法中又调用了 `choose(ILoadBalancer lb, Object key)`这个重载的 `choose` 方法,在这个重载的 `choose` 方法中,每次利用 `random` 对象生成一个不大于服务实例总数的随机数,并将该数作为下标以获取一个服务实例。

➤ RoundRobinRule

`RoundRobinRule` 负载均衡策略叫作线性负载均衡策略,也是负载均衡器中默认的负载均衡策略。这个类的 `choose(ILoadBalancer lb, Object key)`函数整体逻辑是这样的:开启一个计数器 `count`,在 `while` 循环中遍历服务清单,获取清单之前先通过 `incrementAndGetModulo` 方法获取一个下标,这个下标是一个不断自增长的数(先加 1,然后和服务清单总数取模获取到的),拿着下标再去服务清单列表中取服务,每次循环计数器都会加 1,如果连续 10 次都没有取到服务,则会报一个警告 `No available alive servers after 10 tries from load balancer: XXXX`。

➤ RetryRule

带重试功能的负载均衡策略, `RetryRule` 中又定义了一个 `subRule`,它的实现类是 `RoundRobinRule`,然后在 `RetryRule` 的 `choose(ILoadBalancer lb, Object key)`方法中,每次还是采用 `RoundRobinRule` 中的 `choose` 规则来选择一个服务实例,如果选到的实例正常则返回,如果选择的服务实例为 `null` 或者已经失效,则在失效时间 `deadline` 之前不断地进行重试,如果超过了 `deadline` 还是没取到则会返回一个 `null`。

➤ WeightedResponseTimeRule

`WeightedResponseTimeRule` 是 `RoundRobinRule` 的一个子类,在 `WeightedResponseTimeRule` 中对 `RoundRobinRule` 的功能进行了扩展, `WeightedResponseTimeRule` 中会根据每一个实例的运行情况来计算出该实例的一个权重,然后在挑选实例时根据权重进行挑选,这样能够实现更优的实例调用。

➤ BestAvailableRule

BestAvailableRule 继承自 ClientConfigEnabledRoundRobinRule, ClientConfigEnabledRoundRobinRule 内部定义了 RoundRobinRule, 还是采用了 RoundRobinRule 的 choose 方法, 所以它的选择策略和 RoundRobinRule 的选择策略一致。它在 ClientConfigEnabledRoundRobinRule 的基础上主要增加了根据 loadBalancerStats 中保存的服务实例的状态信息来过滤掉失效的服务实例的功能, 然后顺便找出并发请求最小的服务实例来使用。然而 loadBalancerStats 有可能为 null, 如果 loadBalancerStats 为 null, 则 BestAvailableRule 将采用它的父类即 ClientConfigEnabledRoundRobinRule 的服务选取策略。

➤ PredicateBasedRule

PredicateBasedRule 是 ClientConfigEnabledRoundRobinRule 的一个子类, 它先通过内部定义的一个过滤器过滤出一部分服务实例清单, 然后采用线性轮询的方式从过滤出来的结果中选取一个服务实例。

➤ ZoneAvoidanceRule

ZoneAvoidanceRule 是 PredicateBasedRule 的一个实现类, 只不过这里多了一个过滤条件, ZoneAvoidanceRule 中的过滤条件是以 ZoneAvoidancePredicate 为主过滤条件和以 AvailabilityPredicate 为次过滤条件组成的一个叫作 CompositePredicate 的组合过滤条件, 过滤成功之后, 继续采用线性轮询的方式从过滤结果中选择一个出来。

5.2.2 Ribbon

介绍

Ribbon 是 Netflix 发布的云中间层服务开源项目, 其主要功能是提供客户侧软件负载均衡算法, 将 Netflix 的中间层服务连接在一起。

Ribbon 的架构和原理如图 5-7 所示。

Ribbon 的工作分为两步:

(1) 优先选择 Eureka Server, 它优先选择在同一个 Zone 且负载较少的 Server。

(2) 根据用户指定的策略, 从 Server 取到的服务注册列表选择一个地址。其中 Ribbon 提供了多重策略, 例如, 轮询 round robin、随机 Random、根据相应时间加权等。

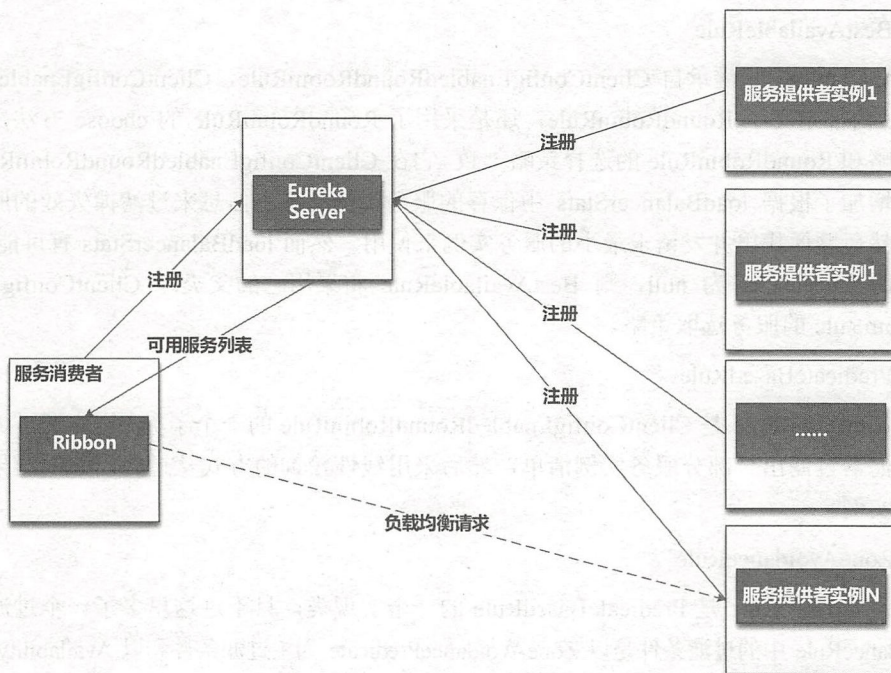


图 5-7 Ribbon 原理

实现

(1) 新建工程 chapter-05-02-ribbon，并且加入 Ribbon 依赖，同时也要加入 Eureka 客户端依赖。

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>Spring-cloud-starter-ribbon</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>Spring-cloud-starter-eureka</artifactId>
</dependency>
```

(2) 配置文件中，程序名称为 service-ribbon，程序端口为 8764，并且指定注册中心地址。

```
eureka:
  client:
    serviceUrl:
```



```

        defaultZone: http://localhost:8761/eureka/
server:
  port: 8764
Spring:
  application:
    name: service-ribbon

```

(3) 在应用启动类上添加@EnableDiscoveryClient，并且注入 bean restTemplate。

```

@Bean
@LoadBalanced
RestTemplate restTemplate() {
    return new RestTemplate();
}

```

@LoadBalanced 注解会自动构造 LoadBalancerClient 接口的实现类并注册到 Spring 容器中。接下来使用 RestTemplate 进行 REST 操作时，会自动使用负载均衡策略，它内部会在 RestTemplate 中加入 LoadBalancerInterceptor 拦截器，这个拦截器的作用就是使用负载均衡。

(4) 新建 controller 类 RibbonController，添加如下代码。

```

@Autowired
private RestTemplate restTemplate;

@GetMapping(value = "/add")
public String add() {
    return restTemplate.getForEntity("http://CLOUD-CLIENT/add?a=30&b=50",
String.class).getBody();
}

```

使用服务注册中心指定的名称调用相应的服务。

测试

(1) 启动注册中心 Eureka Server。

(2) 复制工程 chapter-05-01-eureka-registry，工程结构与其相同，将端口改为 8081 并启动。这时在注册中心应该能看到两个相同名称的实现，分别是 8080 和 8081。

(3) 启动 Ribbon 客户端工程。注册中心中能够看到 Ribbon 客户端服务。

(4) 打开浏览器输入 http://localhost:8764/add，多输入几次，就可以看到两个服务提供者分别输出的日志。

chapter-05-01-eurekaregistry 的日志为:

```
2017-11-21 15:15:46.459 INFO 10336 --- [nio-8080-exec-3]
c.c.s.c.ServiceInstanceRestController : chapter-05-01-eurekaregistry /add,
host:DESKTOP-E3I9LR5, service_id:cloud-client, result:80
2017-11-21 15:15:50.058 INFO 10336 --- [nio-8080-exec-4]
c.c.s.c.ServiceInstanceRestController : chapter-05-01-eurekaregistry /add,
host:DESKTOP-E3I9LR5, service_id:cloud-client, result:80
```

chapter-05-01-eurekaregistry2 的日志为:

```
2017-11-21 15:15:48.686 INFO 5364 --- [nio-8081-exec-1]
c.c.s.c.ServiceInstanceRestController : chapter-05-01-eurekaregistry2 /add,
host:DESKTOP-E3I9LR5, service_id:cloud-client, result:80
2017-11-21 15:15:51.298 INFO 5364 --- [nio-8081-exec-2]
c.c.s.c.ServiceInstanceRestController : chapter-05-01-eurekaregistry2 /add,
host:DESKTOP-E3I9LR5, service_id:cloud-client, result:80
```

每次请求分别由不同的服务提供者提供服务。

总结

在软件的服务层, 基于微服务, 我们可以根据自己微服务的使用情况, 将一些使用频率较高的服务进行横向扩展, 并且基于一定的负载均衡算法, 完成软件的负载均衡。

5.2.3 Feign

介绍

在 Spring Cloud Netflix 中, 各个微服务都是以 HTTP 接口的形式暴露自身服务的, 因此在调用远程服务时必须使用 HTTP 客户端。我们可以使用 JDK 原生的 URLConnection、Apache 的 Http Client、Netty 的异步 HTTP Client, 以及 Spring 的 RestTemplate。但是, 用起来最方便、最优雅的还是 Feign。

Feign 是一种声明式、模板化的 HTTP 客户端。在 Spring Cloud 中使用 Feign, 我们可以做到使用 HTTP 请求远程服务时能与调用本地方法一样的编码体验, 开发者完全感知不到这是远程方法, 更感知不到这是个 HTTP 请求。

为了让 Feign 知道在调用方法时应该向哪个地址发请求, 以及请求需要带哪些参数, 我们需要定义一个接口, 代码如下:


```

@FeignClient(name = "ea")
public interface AdvertGroupRemoteService {

    @RequestMapping(value = "/group/{groupId}", method = RequestMethod.GET)
    AdvertGroupVO findByGroupId(@PathVariable("groupId") Integer adGroupId)

    @RequestMapping(value = "/group/{groupId}", method = RequestMethod.PUT)
    void update(@PathVariable("groupId") Integer groupId, @RequestParam
("groupName") String groupName)

```

- `@FeignClient` 用于通知 Feign 组件对该接口进行代理，使用者可直接通过 `@Autowired` 注入。
- `@RequestMapping` 表示在调用该方法时需要向 `/group/{groupId}` 发送 GET 请求。
- `@PathVariable` 用来对指定请求的 URL 路径里面的变量。

Spring Cloud 应用在启动时，Feign 会扫描标有 `@FeignClient` 注解的接口，生成代理，并注册到 Spring 容器中。生成代理时 Feign 会为每个接口方法创建一个 `RequestTemplate` 对象，该对象封装了 HTTP 请求需要的全部信息，请求参数名、请求方法等信息都是在这个过程中确定的，Feign 的模板化就体现在这里。

Feign 支持可插拔的编码器和解码器。Feign 默认集成了 Ribbon，并和 Eureka 结合，默认实现了负载均衡的效果。

Feign 将方法签名中方法参数对象序列化为请求参数放到 HTTP 请求中的过程，是由编码器（Encoder）完成的。同理，将 HTTP 响应数据反序列化为 Java 对象是由解码器（Decoder）完成的。默认情况下，Feign 会将标有 `@RequestParam` 注解的参数转换成字符串添加到 URL 中，将没有注解的参数通过 Jackson 转换成 JSON 放到请求体中。如果在 `@RequestMapping` 中的 `method` 将请求方式指定为 POST，那么所有未标注的参数会被忽略，示例代码如下：

```

@RequestMapping(value = "/group/{groupId}", method = RequestMethod.GET)
void update(@PathVariable("groupId") Integer groupId, @RequestParam
("groupName") String groupName, DataObject obj);

```

此时因为声明的是 GET 请求，没有请求体，所以 `obj` 参数就会被忽略。

在 Spring Cloud 环境下，Feign 的 `Encoder*` 只会用来编码没有添加注解的参数*。如果自定义了 `Encoder`，那么只有在编码 `obj` 参数时才会调用 `Encoder`。对于 `Decoder`，默认会委托给 Spring MVC 中的 `MappingJackson2HttpMessageConverter` 类进行解码。只有当状态码不在 200~300 之

间时 `ErrorDecoder` 才会被调用。`ErrorDecoder` 的作用是根据 HTTP 响应信息返回一个异常，该异常可以在调用 Feign 接口的地方被捕获到。

实现

(1) 新建工程 `chapter-05-02-feign`，并且添加 Feign 依赖。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>Spring-cloud-starter-feign</artifactId>
</dependency>
```

(2) 添加启动服务发现和启动 Feign 功能。添加如下注释：

```
@EnableDiscoveryClient //用于启动服务发现功能
@EnableFeignClients //用于启动 Feign 功能
```

(3) 定义远程调用接口。创建 `CloudClient` 接口，使用 `@FeignClient("CLOUD-CLIENT")` 注解修饰，`CLOUD-CLIENT` 就是服务提供者的名称，然后定义要使用的服务代码如下：

```
@FeignClient("CLOUD-CLIENT")
public interface CloudClient {
    @RequestMapping(method = RequestMethod.GET, value = "/add")
    Integer add(@RequestParam(value = "a") Integer a, @RequestParam(value = "b") Integer b);
}
```

`@FeignClient` 用于通知 Feign 组件对该接口进行代理（不需要编写接口实现），使用者可直接通过 `@Autowired` 注入。

(4) 新建控制层类 `FeignController`，并且注入 `CloudClient` 接口。代码如下：

```
@Autowired
private CloudClient cloudClient;
@GetMapping(value = "/add")
public Integer add() {
    return cloudClient.add(20, 40);
}
```

(5) 添加配置文件如下：


```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
  server:
    port: 8765
Spring:
  application:
    name: service-feign
```

测试

- (1) 启动注册中心和两个服务提供者。
- (2) 启动 Feign 客户端，并且能在注册中心看到两个服务提供者和 SERVICE-FEIGN 消费者。
- (3) 打开浏览器，输入 `http://localhost:8765/add`，能看到服务提供者的日志交替输出，结果与上一节的结果相同。

总结

通过 Feign，我们能把 HTTP 远程调用对开发者完全透明，得到与调用本地方法一致的编码体验。如果用 Spring Cloud Netflix 搭建微服务，那么 Feign 无疑是最佳选择。

5.2.4 加入 core

介绍

在实际项目中，都是使用声明式调用服务，而不会在客服端和服务端存储两份相同的模型和接口定义。Feign 在 RestTemplate 的基础上对其进行封装，由它来帮助我们定义和实现依赖服务接口的定义。Spring Cloud Feign 是基于 Netflix Feign 实现的，整合 Spring Cloud Ribbon 与 Spring Cloud Hystrix，并且实现了声明式的 Web 服务客户端定义方式。

实现

- (1) 新建工程 `chapter-05-02-core`，因为是依赖工程，所以需要打包成 jar 包。

```
<groupId>cloudskyme</groupId>
<artifactId>core</artifactId>
<version>0.0.1</version>
```

```
<packaging>jar</packaging>
```

(2) 新建工程接口 `ServiceRemoteApi`，然后新建要传输的对象 `User`，实现 `Serializable`。接口中有三个定义，用于远程实现。如果是在本地测试，则执行 `mvn clean install`，这样就能在本地使用相应的 `jar` 文件了，如果是多人协作，则需要建立私服，一般推荐使用 `Nexus`，上传到私服后下载使用。

```
@RequestMapping("/service-remote")
public interface ServiceRemoteApi {

    @RequestMapping(value = "/getName", method = RequestMethod.GET)
    String getName(@RequestParam("name") String name);

    @RequestMapping(value = "/getNameAndAge", method = RequestMethod.GET)
    User getNameAndAge(@RequestHeader("name") String name, @RequestHeader("age") Integer age);

    @RequestMapping(value = "/getUser", method = RequestMethod.POST)
    String getUser(@RequestBody User user);
}
```

(3) 在 `chapter-05-01-eureka-registry` 服务提供者的工程中添加对以上包的依赖，并且添加实现。

```
public String getName(@RequestParam("name") String name)
public User getNameAndAge(@RequestHeader("name") String name,
    @RequestHeader("age") Integer age)
public String getUser(@RequestBody User user)
```

需要注意的是，方法中的参数 `@RequestHeader` 和 `@RequestBody` 注解还是要添加的，`@RequestParam` 注解可以不添加。

(4) 在消费端工程中加入包依赖，并且将调用接口继承自此接口，代码如下：

```
@FeignClient("CLOUD-CLIENT")
public interface BackgroundService extends ServiceRemoteApi{
}
```

(5) 添加测试方法调用远程接口实现的方法。


```

@RequestMapping("/feign")
public Map<String, Object> feignCore() {
    Map<String, Object> ret = new HashMap<String, Object>();
    StringBuffer sb = new StringBuffer();
    String s1 = backgroundService.getName("张三");
    sb.append(s1).append("\n");
    User user = null;
    try {
        user = backgroundService.getNameAndAge(URLEncoder.encode("李四", "UTF-8"), 20);
    } catch (UnsupportedEncodingException e) {
        e.printStackTrace();
    }
    sb.append(user.toString()).append("\n");
    String s3 = backgroundService.getUser(new User("王五", 39));
    sb.append(s3).append("\n");
    ret.put("show", sb.toString());
    return ret;
}

```

测试

- (1) 启动注册中心，并且启动服务提供者。
- (2) 启动 Feign 工程，能够在注册中心看到注册的服务。
- (3) 打开浏览器，输入 <http://localhost:8765/feign>，在服务提供者工程中能够看到日志输出：

```

2017-11-23 13:11:39.152 INFO 9848 --- [nio-8080-exec-3]
c.c.s.controller.ServiceRemoteApiImpl : chapter-05-01-eurekaregistry 张三
2017-11-23 13:11:39.165 INFO 9848 --- [nio-8080-exec-4]
c.c.s.controller.ServiceRemoteApiImpl : chapter-05-01-eurekaregistry 李四
2017-11-23 13:11:39.178 INFO 9848 --- [nio-8080-exec-5]
c.c.s.controller.ServiceRemoteApiImpl : chapter-05-01-eurekaregistry
User{name='王五', age=39}

```

并且能够看到输出结果。

总结

通过上面的介绍，大家应该已经体会到了 Feign 继承特性的方便之处了，这种方式用起来

确实很方便，但是也带来一个问题，就是服务提供者和服务消费者的耦合度太高，此时如果服务提供者修改了一个接口的定义，服务消费者可能也得跟着变化，进而带来很多未知的工作量，因此在使用继承特性时，要慎重考虑使用的场景和方式。

5.3 微服务容错（Hystrix）

在大中型分布式系统中，通常系统有很多依赖。在并发量很小的时候，通常不会造成很严重的后果，但是当并发量激增，这些依赖的稳定性就有可能造成整个系统的瘫痪，这也就是我们经常说的雪崩。

5.3.1 雪崩的形成

服务雪崩效应是一种因服务提供者的不可用而导致服务调用者的不可用的现象，并将不可用逐渐放大的过程。举例来说，我们使用链式设计模式构建的微服务，当其他的服务出现问题时，就会出现连锁支应，导致整个服务链条不可用。

造成服务不可用的原因包括：

- 硬件故障；
- 网络连接缓慢；
- 程序 Bug；
- 缓存击穿，一般发生在缓存应用重启，所有缓存被清空时，以及短时间内大量缓存失效时。大量的缓存不命中，使请求直击后端，造成服务提供者超负荷运行，引起服务不可用；
- 用户大量请求。

那么出现雪崩应该如何应对呢？

5.3.2 应对方案

应对雪崩一般有以下几种办法。

➤ 流量控制

一般是使用 Nginx 进行流量控制，对流量大的应用采用分流和限制，这个功能也可以使用网关来完成。

➤ 服务自动扩容

取决于硬件的限制，也可以使用第三方的云服务以达到扩容的效果。如果微服务构建得比较成熟，则可以通过容器的动态扩容来完成服务的扩容。

➤ 降级和资源隔离

资源隔离主要是对调用服务的线程池进行隔离，监控一般要细致到线程级别，当发现某个线程占用资源过高时，进行有效的处理来解决性能瓶颈。

我们根据具体业务将依赖服务分为强依赖和弱依赖。强依赖服务不可用会导致当前业务中止，而弱依赖服务的不可用不会导致当前业务的中止。

不可用服务的调用快速失败一般通过超时机制、熔断器和熔断后的降级方法来实现。

5.3.3 降级和熔断

➤ 降级

在网络访问中，为了优化用户体验，遇到超时的情况，可以直接放弃本次请求，不等待结果的返回，直接返回用户默认数据，也可以降级为从另一个服务或者使用缓存中设置的默认数据。

➤ 熔断

熔断是指错误达到某个设定的阈值，或者请求量超过阈值后，系统自动（或手动）阻止代码或服务的执行调用，从而达到系统整体保护的效果。当检测到系统可用时，需要恢复访问。

熔断器模式

熔断器模式定义了熔断器开关相互转换的逻辑。

➤ 正常运行（Closed）

当一个系统运行平稳时，成功状态计数器用于测量弹性系统的稳定性，而故障表用于跟踪任何故障。该设计确保当达到故障的阈值时，断路器断开电路，以防止进一步的资源请求。

➤ 失败状态（Open）

在这个时刻，每一个依赖调用是短路的，并抛出 `HystrixRuntimeException` 异常，伴随 `SHORTCIRCUIT` 失败类型，给出异常明确的原因。一旦等待时间过后，`Hystrix` 断路器移到半开放状态。

➤ 半开放状态

在这种状态下，由 `Hystrix` 负责发送第一个请求，检查系统的可用性，让其他的请求快速失败，直到得到依赖的响应。如果调用是成功的，则断路器被重置为 `Closed` 状态；如果发生故障，

则系统返回 Open 状态，并且整个过程继续循环。

断路器是 Hystrix 库默认提供的一个功能。断路器的功能可以概括如下：

- 熔断器对所有调用状态进行验证；
- 如果是 closed 状态，则允许请求通过；
- 如果是 open 状态，则失败所有的请求；
- 如果是 half-open 状态，则允许一个请求通过，并在成功或者失败时，转换成 closed 或 open 状态。

5.3.4 Hystrix

介绍

Hystrix：通过服务熔断、降级、限流、异步 RPC 等手段控制依赖服务的延迟与失败。通过命令模式封装调用来实现弹性保护，继承 `HystrixCommand` 并且实现 `run` 方法，就完成了最简单的封装。可以为分布式服务提供弹性保护。

Hystrix 的设计原则包括：资源隔离、熔断器、命令模式。

➤ 断路器机制

断路器很好理解，当 Hystrix Command 请求后端服务失败数量超过一定比例，默认为 50%，断路器会切换到开路状态（open），这时所有请求会直接失败而不会发送到后端服务。断路器保持在开路状态一段时间后，默认为 5 秒，自动切换到半开路状态（half-open），这时会判断下一次请求的返回情况。如果请求成功，则断路器切回闭路状态（closed），否则重新切换到开路状态（open）。Hystrix 的断路器就像我们家庭电路中的保险丝，一旦后端服务不可用，断路器会直接切断请求链，避免发送大量无效请求影响系统吞吐量，并且断路器有自我检测并恢复的能力。

➤ fallback

fallback 相当于降级操作。对于查询操作，我们可以实现一个 fallback 方法，当请求后端服务出现异常时，可以使用 fallback 方法返回的值。fallback 方法的返回值一般是设置的默认值或者来自缓存。

➤ 资源隔离

在 Hystrix 中，主要通过线程池来实现资源隔离。通常在使用时我们会根据调用的远程服务划分出多个线程池。例如，调用产品服务的 Command 放入 A 线程池，调用账户服务的 Command 放入 B 线程池。这样做的主要优点是运行环境被隔离开了，就算调用服务的代码存在 Bug 或者由于其他原因导致自己所在线程池被耗尽时，不会对系统的其他服务造成影响。但代价就是维护多个线程池会对系统带来额外的性能开销。如果对性能有严格要求而且确信自己调用服务的

客户端代码不会出问题，则可以使用 Hystrix 的信号模式（Semaphores）来隔离资源。

Hystrix 服务调用的内部逻辑如图 5-8 所示。

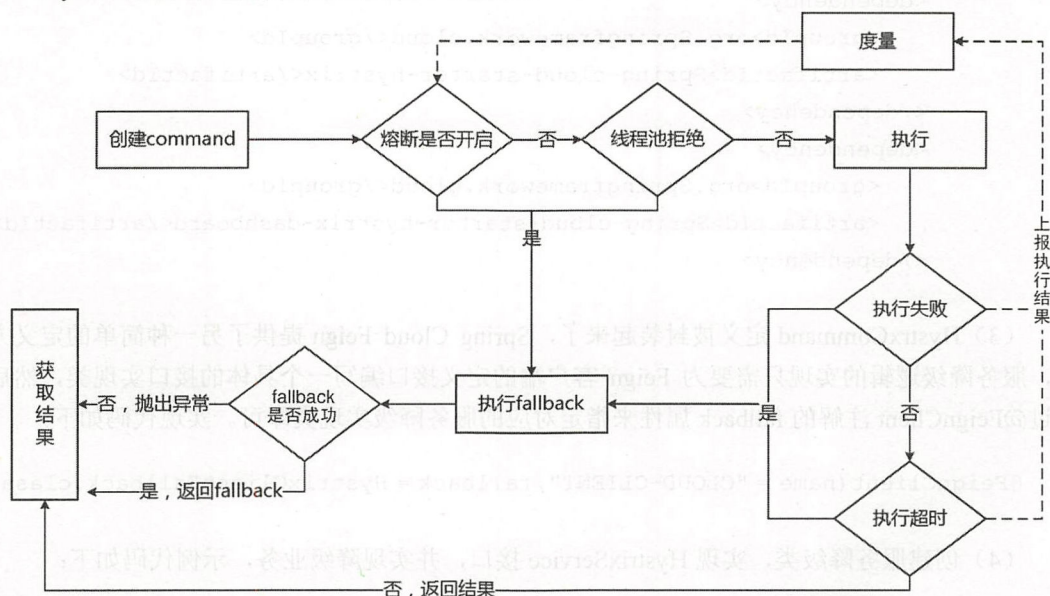


图 5-8 Hystrix 服务调用流程

- 构建的 Command 对象，调用执行方法。
- Hystrix 检查当前服务的熔断器开关是否开启，若开启，则执行降级服务 getFallback 方法。
- 若熔断器开关关闭，则 Hystrix 检查当前服务的线程池是否能接收新的请求，若超过线程池已满，则执行降级服务 getFallback 方法。
- 若线程池接收请求，则 Hystrix 开始执行服务调用具体逻辑 run 方法。
- 若服务执行失败，则执行降级服务 getFallback 方法，并将执行结果上报 Metrics 更新服务健康状况。
- 若服务执行超时，则执行降级服务 getFallback 方法，并将执行结果上报 Metrics 更新服务健康状况。
- 若服务执行成功，则返回正常结果。
- 若服务降级方法 getFallback 执行成功，则返回降级结果。
- 若服务降级方法 getFallback 执行失败，则抛出异常。

实现

(1) 新建工程 chapter-05-03-03，为了演示断路器的功能，我们继续上一节使用的 Feign 示例。

(2) 新加入 Hystrix 依赖。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>Spring-cloud-starter-hystrix</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>Spring-cloud-starter-hystrix-dashboard</artifactId>
</dependency>
```

(3) HystrixCommand 定义被封装起来了, Spring Cloud Feign 提供了另一种简单的定义方式, 服务降级逻辑的实现只需要为 Feign 客户端的定义接口编写一个具体的接口实现类, 然后通过 @FeignClient 注解的 fallback 属性来指定对应的服务降级实现类即可。实现代码如下:

```
@FeignClient(name = "CLOUD-CLIENT", fallback = HystrixClientFallback.class)
```

(4) 创建服务降级类, 实现 HystrixService 接口, 并实现降级业务, 示例代码如下:

```
@Component
static class HystrixClientFallback implements HystrixService
```

(5) 修改配置文件, 设置打开 hystrix 支持, 代码如下:

```
feign:
  hystrix:
    enabled: true
```

(6) 在应用启动类上加入 Hystrix 支持, 同时也提供了 Hystrix Dashboard 的整合, 代码如下:

```
@SpringBootApplication
@EnableFeignClients
@EnableDiscoveryClient
@EnableHystrix
@EnableHystrixDashboard
public class Chapter0503Application {
```

```
    public static void main(String[] args) {
```



```

        new SpringApplicationBuilder(Chapter0503Application.class).
web(true).run(args);
    }
}

```

(7) 在 pom.xml 中加入以下代码。

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>Spring-boot-starter-actuator</artifactId>
</dependency>

```

测试

(1) 启动注册中心 chapter-05-01-eureka-server。

(2) 启动提供者服务 chapter-05-01-eureka-registry。

(3) 启动熔断器工程 chapter-05-03-03，这时在注册中心可以看到 CLOUD-CLIENT 和 SERVICE-FEIGN-HYSTRIX。

(4) 打开浏览器，输入 <http://localhost:8766/feign>，可以看到能够正常访问服务。

(5) 把提供者服务 chapter-05-01-eureka-registry 关闭。查看注册中心，只能看到 SERVICE-FEIGN-HYSTRIX 服务。再次调用刚才的服务。因为服务不能提供，将触发熔断，控制台输出：

```

2017-11-28 15:58:43.066 INFO 12320 --- [ HystrixTimer-3]
s.s.HystrixService$HystrixClientFallback : fail getName(String name)
2017-11-28 15:58:44.070 INFO 12320 --- [ HystrixTimer-1]
s.s.HystrixService$HystrixClientFallback : fail getNameAndAge(String name,
Integer age)
2017-11-28 15:58:45.071 INFO 12320 --- [ HystrixTimer-4]
s.s.HystrixService$HystrixClientFallback : fail getUser(User user)

```

页面上打印出：

```

{"show":"fail getName(String name)\nUser{name='default username',
age=0}\nfail getUser(User user)\n"}

```

(6) 输入 <http://localhost:8766/hystrix>，可以看到如图 5-9 所示的界面。



图 5-9 Hystrix 面板

输入要监控的项目，在这里输入 `http://localhost:8766/Hystrix.stream`，并且输入 Title，同时调用接口，并且重复有提供者和无提供者的测试，可以查看到监控的结果。每个接口的调用情况和性能情况都会进行实时的监控。

总结

Hystrix 不仅仅是一个断路器，也是一个具有丰富监控功能的完整的库，可以很容易地植入现有系统中。我们已经开始为未来的使用情况探索，使用该库的请求崩溃和请求缓存的功能。当然，还有一些其他的 Java 实现，如 Akka 和 Spring 断路器。

5.3.5 集中监控

介绍

dashboard 可以对 Hystrix.stream 的节点监控进行图形化显示。但是 dashboard 一次只能监控一个节点，我们的微服务可能是成百上千的，好在 Apache 提供了 Turbine，Spring Cloud 也对它进行了整合，通过 Turbine 就可以监控集群。

Turbine 是聚合服务器发送事件流数据的一个工具，Hystrix 的监控中，只能监控单个节点，实际生产中都为集群，因此可以通过 Turbine 来监控集群下 Hystrix 的 metrics 情况，通过 Eureka 来发现 Hystrix 服务。

介绍

(1) 新建工程 chapter-05-04，加入 Turbine 支持。

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>Spring-cloud-starter-turbine</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>Spring-cloud-netflix-turbine</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>Spring-boot-starter-actuator</artifactId>
</dependency>
```

(2) 在主方法上添加注解@EnableTurbine。

```
@SpringBootApplication
@EnableTurbine
public class Chapter050304Application {

    public static void main(String[] args) {
        new SpringApplicationBuilder(Chapter050304Application.class)
            .web(true).run(args);
    }
}
```

(3) 在配置文件中添加配置。

```
Spring:
  application.name: service-turbine
server:
  port: 9000
security.basic.enabled: false
turbine:
  aggregator:
    clusterConfig: default # 指定聚合哪些集群，多个使用","分割，默认为 default
```

可使用 `http://.../turbine.stream?cluster={clusterConfig 之一}` 访问

```

appConfig: service-feign-hystrix,service-feign,cloud-client   ### 配置
Eureka 中的 serviceId 列表, 表明监控哪些服务
clusterNameExpression: new String("default")
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/

```

测试

- (1) 启动注册中心 `chapter-05-01-eureka-server`。
- (2) 启动 `chapter-05-03-03`, Hystrix 服务。
- (3) 启动 `chapter-05-03-04`, 输入 `http://localhost:9000/turbine.stream`, 能够看到输出收集的信息。
- (4) 输入 `http://localhost:8766/Hystrix`, 然后在监控的窗口输入 `http://localhost:9000/turbine.stream`, 加入名称, 就能够看到所有服务及接口的调用情况。

总结

在大规模集群的情况下, 使用集中监控是很有必要的。当请求出现异常情况时, 能够第一时间定位原因, 当请求量超过计划值时, 可以动态调整资源进行横向扩展。

5.4 分布式配置中心

微服务化之后, 应用的数量剧增, 临时需要调整配置参数时, 无论是运维直接在服务器上修改, 还是工程中修改配置后重新打包部署, 对运维来说工作量都是巨大的, 而且人为的操作会加大出错的概率, 外化和中心化配置则可以更好地解决分布式环境的配置问题。

介绍

Spring Cloud Config 是 Spring Cloud 团队创建的一个全新项目, 用来为分布式系统中的基础设施和微服务应用提供集中化的外部配置支持, 它分为服务端与客户端两个部分。其中服务端也称为分布式配置中心, 它是一个独立的微服务应用, 用来连接配置仓库并为客户端提供获取配置信息、加密/解密信息等访问接口; 而客户端则是微服务架构中的各个微服务应用或基础设施, 它们通过指定的配置中心来管理应用资源与业务相关的配置内容, 并在启动时从配置中心获取和加载配置信息。Spring Cloud Config 实现了对服务端和客户端中环境变量和属性配置的

抽象映射，所以它除了适用于 Spring 构建的应用程序，还可以在任何其他语言运行的应用程序中使用。Spring Cloud Config 实现的配置中心默认采用 Git 来存储配置信息，Spring Cloud 提供了 2 种方式的外部配置。

- Spring Cloud Config: 通过本地文件系统、Git/SVN 仓库来管理配置文件，可以满足基本外化需求，但不能精细地管理配置项。
- Spring Cloud Zookeeper Config: 通过 ZooKeeper 分级命名空间来储存配置项数据，并且支持基础上下文和 profile 命名空间，另外 ZooKeeper 可以实时监听节点变化和通知机制，应该是首选。

分布式配置中心的架构如图 5-10 所示。

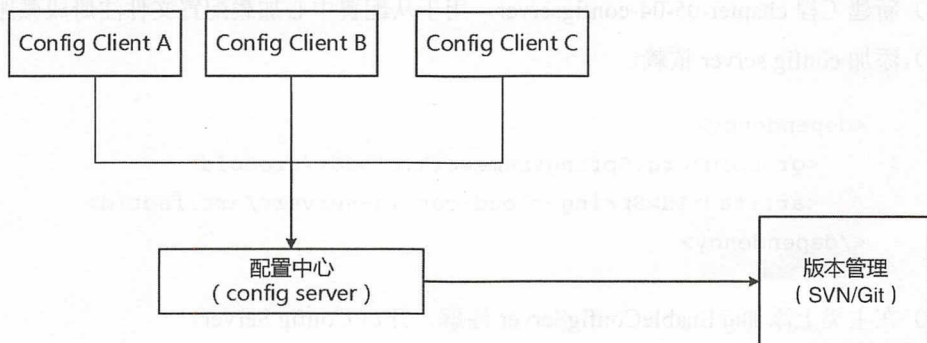


图 5-10 分布式配置中心

图 5-10 简要描述了一个普通 Spring Cloud Config 应用的场景。其中主要有以下几个组件。

➤ Config Client

Client 很好理解，就是使用了 Spring Cloud Config 的应用。

Spring Cloud Config 提供了基于 Spring 的客户端，应用只要在代码中引入 Spring Cloud Config Client 的 jar 包即可工作。

➤ Config Server

Config Server 是需要独立部署的一个 Web 应用，它负责把 SVN/Git 上的配置返回给客户端。

➤ SVN/Git Repository

远程 SVN/Git 仓库，一般而言，我们会把配置放在一个远程仓库，通过现成的 SVN/Git 客户端来管理配置。

Config Server 接收到来自客户端的配置获取请求后，会先把远程仓库的配置“clone”到本地的临时目录，然后从临时目录读取配置并返回。

实现

(1) 新建工程 Spring-cloud-config，用于模拟配置管理服务器上的配置文件。

(2) 新建四个文件，用于模拟开发、测试和生产环境。

- cloudskyme-dev.properties
- cloudskyme-prod.properties
- cloudskyme-test.properties
- cloudskyme.properties

每个文件中都有一个 configvalue 的值，每个文件分别设置不同的值。

(3) 新建工程 chapter-05-04-configserver，用于从配置中心加载配置文件注册成微服务。

(4) 添加 config server 依赖。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>Spring-cloud-config-server</artifactId>
</dependency>
```

(5) 在主类上添加@EnableConfigServer 注解，开启 Config Server。

(6) 添加配置文件，在配置文件中指向工程的配置中心地址。

```
Spring.application.name=service-configserver
server.port=9001
```

```
Spring.cloud.config.server.git.uri=https://github.com/cloudskyme/Spring-
cloud-config
```

```
Spring.cloud.config.server.git.searchPaths=Spring-cloud-config
Spring.cloud.config.label=master
Spring.cloud.config.server.git.username=
Spring.cloud.config.server.git.password=
```

(7) 新建工程 chapter-05-04-configclient，并且加入 config client 配置。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>Spring-cloud-starter-config</artifactId>
</dependency>
```



```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>Spring-boot-starter-actuator</artifactId>
</dependency>
```

(8) 在配置文件中指定 Config Server 的地址，配置文件内容如下：

```
Spring.application.name=cloudskyme
Spring.cloud.config.label=master
Spring.cloud.config.profile=dev
Spring.cloud.config.uri= http://localhost:9001/
server.port=9002
```

(9) 添加控制层，用于取得配置中心的数据，代码如下：

```
@Value("${configvalue}")
String configvalue;
/**
 * @Description 取得配置中心的值
 * @return
 */
@RequestMapping(value = "/getvalue")
public String getConfigvalue(){
    return this.configvalue;
}
```

并且在需要刷新的类上加上@RefreshScope 注解。这样在客户端执行/refresh 时就会更新此类下面的变量值。

测试

(1) 启动 chapter-05-04-configserver 应用，打开浏览器，输入 http://localhost:9001/service-configserver/prod，会得到如下输出：

```
{ "name": "service-configserver", "profiles": ["prod"], "label": null, "version":
"714d884cf2b1dce9d83745db059339ec3862988c", "state": null, "propertySources": [] }
```

URL 与配置文件的映射关系如下：

- /{application}/{profile}/{label}]

- `/{{application}}-{{profile}}.yml`
- `/{{label}}/{{application}}-{{profile}}.yml`
- `/{{application}}-{{profile}}.properties`
- `/{{label}}/{{application}}-{{profile}}.properties`

上面的 URL 会映射 `{{application}}-{{profile}}.properties` 对应的配置文件，`{{label}}` 对应 Git 上不同的分支，默认为 `master`。

(2) 启动 `chapter-05-04-configclient`，启动时会先执行：

```
2017-11-30 17:44:12.361 INFO 16272 --- [           main]
c.c.c.ConfigServicePropertySourceLocator : Fetching config from server at:
http://localhost:9001/
2017-11-30 17:44:17.144 INFO 16272 --- [           main]
c.c.c.ConfigServicePropertySourceLocator : Located environment:
name=cloudskyme, profiles=[dev], label=master,
version=714d884cf2b1dce9d83745db059339ec3862988c, state=null
2017-11-30 17:44:17.145 INFO 16272 --- [           main]
b.c.PropertySourceBootstrapConfiguration : Located property source:
CompositePropertySource [name='configService',
propertySources=[MapPropertySource {name='configClient'}, MapPropertySource
{name='https://github.com/cloudskyme/Spring-cloud-config/Spring-cloud-config
/cloudskyme-dev.properties'}, MapPropertySource
{name='https://github.com/cloudskyme/Spring-cloud-config/Spring-cloud-config
/cloudskyme.properties'}]]
```

在服务端的应用会看到获取配置中心的配置信息。打开浏览器请求 `http://localhost:9002/getvalue`，得到服务器端的配置：

```
dev-environment
```

(3) 修改 `cloudskyme-dev.properties` 中配置文件的内容为 `cloudskyme-dev.properties` 并上传，然后以 POST 方式发送请求 `http://localhost:9002/refresh`，再次用浏览器请求 `http://localhost:9002/getvalue`，就可以得到更新之后的值。

总结

在线上环境，我们一般使用配置中心集群来达到高可用的效果，在应用配置更改时，我们也希望能够自动更新到应用中。

使用 webhook 自动更新，一般的自动发布系统都会支持 webhook，当配置文件的内容发生变化时，会自动通过消息系统将变化的内容通知给客户端，完成配置信息的更新。

5.5 API 网关

在微服务架构中，后端服务往往不直接开放给调用端，而是通过一个 API 网关根据请求的 URL 路由到相应的服务。当添加 API 网关后，在第三方调用端和服务提供方之间就创建了一面墙，这面墙直接与调用方通信进行权限控制，然后将请求均衡分发给后台服务端。

随着微服务架构体系的建立，API 网关扮演着独木桥的角色，其出现在企业系统的边界，承担着企业内部与企业外部交互通信的作用，它除了需要保证数据交换，对接入客户端身份进行认证，防止数据篡改等业务鉴权的功能，还承担了流控、协议转换等作用。

熟悉设计模式的读者一眼就能够看出来，网关其实就是门面模式的一种升级改造，但是它具有更多的功能和特性。门面模式共有两种角色：门面角色和子系统角色。门面模式的类图如图 5-11 所示。

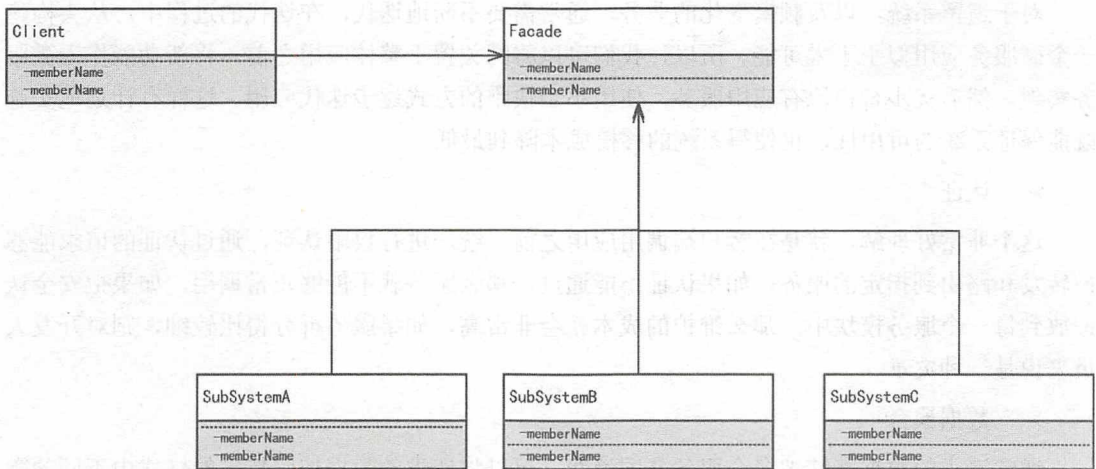


图 5-11 门面模式

门面角色：客户端调用这个方法。此角色知晓相关的子系统的功能和责任。正常情况下，本角色会将所有从客户端发来的请求委派到相应的子系统中。

子系统角色：可以同时有一个或者多个子系统。每个子系统都不是一个单独的类，而是一个类的集合。每一个子系统都可以被客户端直接调用，或者被门面角色直接调用。子系统并不知道门面的存在，对于子系统而言，门面仅仅是另一个客户端而已。

5.5.1 为什么需要网关

网关可以看作一个各种微服务的入口。通常一个系统可以有一个或者多个网关。

可以给不同的端设置不同的访问网关,这样当某个端出现问题时,不会导致整个系统瘫痪。网页端使用 Web 网关,手机端使用手机网关,但可以使用同一套服务。在服务中可以以报文的方式对各个端进行标识。

网关的作用如下。

➤ 路由和版本控制

路由非常好理解,类似 Nginx 做反射代理,将指定的路径路由到特定的服务上。同时,也可以对不同版本的 API 进行相应的版本控制,还可以将多个不同服务产生的结果合并统一输出。这种方式特别适合互联网网站的首页或者某个端的功能页面,因为一个页面往往由多个模块组成,每个模块通过网关对特定的服务进行访问,获取需要的内容。

➤ 迭代设计

对于遗留系统,以及频繁变化的业务,通常需要不断地迭代,在迭代的过程中,从头构建一个微服务应用似乎不太可能。所以,我们可以将网关置于整体应用之前,将新功能作为微服务实现,然后逐步替换原有应用服务,使用小步快跑的方式逐步迭代应用。这样有计划地实施既能保证系统的可用性,也使得系统的移植成本降到最低。

➤ 认证

这个非常好理解,就是在客户端调用应用之前,统一进行权限认证,通过认证的请求能够被转发和路由到指定的服务,如果认证不能通过,那么服务就不能够正常调用。如果把安全认证放到每一个服务模块中,那么维护的成本就会非常高,如果服务拆分得比较细,则对开发人员来说是一种灾难。

➤ 数据聚合

通常请求的数据能够被多个服务共同消费,可以将请求的数据按照特定的格式由不同的微服务进行消费。这种场景类似于路由和版本控制的情况,客户端根据需要的内容由网关统一转发,最终将聚合的结果进行展示。

➤ 格式转换

网关能够将不同的数据格式进行统一的转换,比如前端请求发送的是 XML 格式的数据,而微服务需要的是 JSON 格式的数据,那么可以由网关收到数据后进行转换,分别调用不同的微服务进行处理。如果系统中有对接的场景,则这种方式特别适合,对外使用统一的一种格式,而内部根据不同的微服务转换成指定的数据格式。

➤ 协议转换

一般对外提供服务使用的都是 HTTP REST 的方式，可以通过网关转换协议，向不同的微服务进行请求。协议可以是 GRPC，也可以是 HTTP 的方式，当然也可以是 Web Service 的方式。

➤ 限流和缓存

对于访问量比较大的应用，可以适当地对访问的频率和速率进行控制。同时也可以使用缓存的方案，以应用大数据量的请求。有了网关，就可以有针对性地对特定的请求进行二次处理。

➤ 日志记录

对于每个服务的访问次数、访问的返回时长进行统一记录，这样系统中哪些服务有问题，就可以定量地进行评估以进行性能优化。

在实现 API Gateway 时，应当避免将非通用逻辑放入其中，应当尽量脱离业务逻辑。

5.5.2 Zuul

在 Spring Cloud 体系中，Spring Cloud Zuul 就是提供负载均衡、反向代理、权限认证的一个 API Gateway。

开源地址：<https://github.com/Netflix/zuul>。

Spring Cloud Zuul 路由是微服务架构不可或缺的一部分，提供动态路由、监控、弹性、安全等边缘服务。Zuul 是 Netflix 出品的一个基于 JVM 路由和服务端的负载均衡器。

Zuul 提供了四种过滤器的 API，分别为前置（Pre）、后置（Post）、路由（Route）和错误（Error）四种处理方式。

一个请求会先按顺序通过所有的前置过滤器，之后在路由过滤器中转发给后端应用，得到响应后又会通过所有后置过滤器，最后响应给客户端。在整个流程中，如果发生了异常，则会跳转到错误过滤器中。

一般来说，如果需要在请求到达后端应用前就进行处理，则会选择前置过滤器。例如，鉴权、请求转发、增加请求参数等行为。在请求完成后需要处理的操作放在后置过滤器中完成，例如，统计返回值和调用时间、记录日志、增加跨域头等行为。路由过滤器一般只需要选择 Zuul 中内置的即可，错误过滤器一般只需要一个，这样可以在 Gateway 遇到错误逻辑时直接抛出异常中断流程，并直接统一处理返回结果。

最通常的应用就是鉴权，一般来说整个服务的鉴权逻辑可以很复杂。

- 客户端：App、Web、Backend。
- 权限组：用户、后台人员、其他开发者。

- 实现：OAuth、JWT。
- 使用方式：Token、Cookie、SSO。

而对于后端应用来说，它们其实只需要知道请求属于谁，而不需要知道为什么，所以 Gateway 可以友善地帮助后端应用完成鉴权这个行为，并将用户的唯一标示透传到后端，而不需要、甚至不应该将身份信息也传递给后端，防止某些应用利用这些敏感信息做不安全的事情。

Zuul 默认情况下在处理后会删除请求的 Authorization 头和 Set-Cookie 头，也算是贯彻了这个原则。

Zuul 的架构如图 5-12 所示。

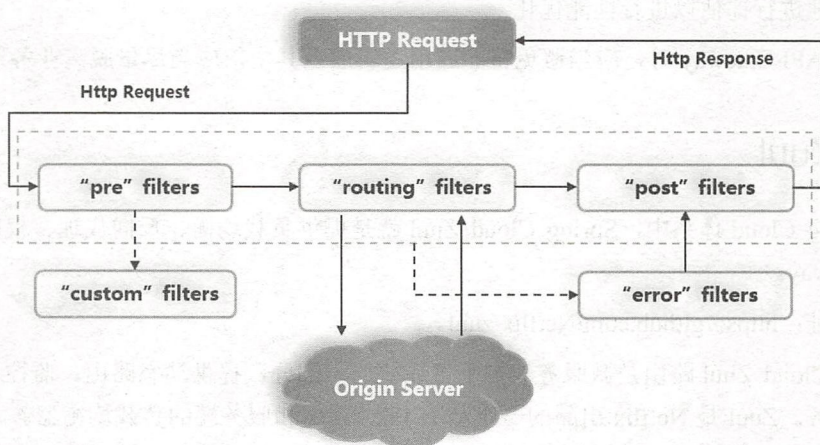


图 5-12 Zuul 的架构

在 Zuul 中，整个请求的过程是这样的，首先将请求交给 `zuulServlet` 处理，`zuulServlet` 中有一个 `zuulRunner` 对象，该对象中初始化了 `RequestContext`：作为存储整个请求的一些数据，并被所有的 `zuulfilter` 共享。`zuulRunner` 中还有 `FilterProcessor`，`FilterProcessor` 作为执行所有的 `zuulfilter` 的管理器。`FilterProcessor` 从 `filterloader` 中获取 `zuulfilter`，而 `zuulfilter` 被 `filterFileManager` 所加载，并支持 Groovy 热加载，采用了轮询的方式热加载。有了这些 filter 之后，`zuulServlet` 首先执行 pre 类型的过滤器，再执行 route 类型的过滤器，最后执行的是 post 类型的过滤器，如果在执行这些过滤器有错误时，则会执行 error 类型的过滤器。执行完这些过滤器，最终将请求的结果返回给客户端。

Servlet 通过一个定义良好的生命周期来进行管理，该生命周期规定了 Servlet 如何被加载、实例化、初始化、处理客户端请求，以及何时结束服务。该生命周期可以通过 `javax.servlet.Servlet` 接口中的 `init`、`service` 和 `destroy` 这些 API 来表示，所有 Servlet 必须直接或间接地实现 `GenericServlet` 或 `HttpServlet` 抽象类。

Servlet 的生命周期有四个阶段：加载并实例化、初始化、请求处理、销毁。主要涉及的方法有 `init`、`service`、`doGet`、`doPost`、`destroy` 等。

Zuul 逻辑简化版的代码如下：

```
public void service(ServletRequest servletRequest, ServletResponse
servletResponse) {
    // 用于初始化 RequestContext
    init((HttpServletRequest) servletRequest, (HttpServletResponse)
servletResponse);
    /* RequestContext 用于记录 Request 的 context。前面也分析了，由于
Servlet 是单例多线程的，而 Request 由唯一 worker 线程处理，这里的 RequestContext 使用
ThreadLocal 实现，其本身简单 wrap 了 ConcurrentHashMap */
    RequestContext context = RequestContext.getCurrentContext();
    // 执行 Pre filters 逻辑
    preRoute();
    // 执行 route 逻辑
    route();
    // 执行 postRoute 逻辑
    postRoute();
}
```

RequestContext 提供了执行 filter Pipeline 所需要的 Context，因为 Servlet 是单例多线程，这就要求 RequestContext 既要线程安全，又要 Request 安全。context 使用 ThreadLocal 保存，这样每个 worker 线程都有一个与其绑定的 RequestContext，因为 worker 仅能同时处理一个 Request，这就保证了 Request Context 既是线程安全的又是 Request 安全的。所谓 Request 安全，即该 Request 的 Context 不会与其他同时处理 Request 冲突。

介绍

Zuul 的功能：

- 认证；
- 压力测试；
- 金丝雀测试；
- 动态路由；
- 负载削减；
- 安全；

- 静态响应处理；
- 主动/主动交换管理。

Zuul 的规则引擎允许通过任何 JVM 语言来编写规则和过滤器，支持基于 Java 和 Groovy 的构建。

实现

(1) 新建工程 chapter-05-05，添加 Zuul 依赖。

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>Spring-cloud-starter-zuul</artifactId>
</dependency>
```

(2) 添加配置，代码如下：

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
server:
  port: 9999
Spring:
  application:
    name: service-zuul
zuul:
  routes:
    api-a:
      path: /api-a/**
      serviceId: service-ribbon
    api-b:
      path: /api-b/**
      serviceId: service-feign
```

首先指定服务注册中心的地址为 `http://localhost:8761/eureka/`，服务的端口为 9999，服务名为 `service-zuul`；以 `/api-a/` 开头的请求都转发给 `service-ribbon` 服务；以 `/api-b/` 开头的请求都转发给 `service-feign` 服务。

(3) 在应用启动类上添加 `@EnableZuulProxy`，开启 Zuul 功能。Zuul 为我们提供了两个主

应用注解：`@EnableZuulServer` 和 `@EnableZuulProxy`，其中 `@EnableZuulProxy` 包含 `@EnableZuulServer` 的功能，而且还加入了 `@EnableCircuitBreaker` 和 `@EnableDiscoveryClient`。当我们需要运行一个没有代理功能的 Zuul 服务，或者有选择地开关部分代理功能时，则需要使用 `@EnableZuulServer` 替代 `@EnableZuulProxy`。这时添加任何 `ZuulFilter` 类型实体类都会被自动加载。

(4) Zuul 不仅有路由功能，并且还能过滤，做一些安全验证。添加类 `PreRequestLogFilter`，代码如下：

```
@Override
public Object run() {
    RequestContext ctx = RequestContext.getCurrentContext();
    HttpServletRequest request = ctx.getRequest();
    PreRequestLogFilter.logger
        .info(String.format("send %s request to %s", request.getMethod(),
request.getRequestURL().toString()));
    return null;
}
```

自定义的 Zuul Filter 需要实现以下几个方法。

- `filterType`: 返回过滤器的类型。有 `pre`、`route`、`post`、`error` 等几种取值，分别对应上文的几种过滤器。详细可以参考 `com.netflix.zuul.ZuulFilter.filterType()` 中的注释。
- `filterOrder`: 返回一个 `int` 值来指定过滤器的执行顺序，不同的过滤器允许返回相同的数字。
- `shouldFilter`: 返回一个 `boolean` 值来判断该过滤器是否要执行，`true` 表示执行，`false` 表示不执行。
- `run`: 过滤器的具体逻辑。本例中，我们让它打印了请求的 HTTP 方法及请求的地址。

(5) 在启动类中添加注入。

```
@Bean
public PreRequestLogFilter preRequestLogFilter() {
    return new PreRequestLogFilter();
}
```

测试

(1) 启动注册中心 `chapter-05-01-eureka-server`。

(2) 启动服务提供者 chapter-05-01-eureka-registry。

(3) 启动 Ribbon 消费端 chapter-05-02-ribbon。

(4) 启动 Feign 消费端 chapter-05-02-feign。

(5) 启动工程 chapter-05-05，能够看到已经启动的服务列表。

在浏览器中输入 `http://localhost:9999/api-a/add`，可以看到返回计算的结果，说明用此路由已经路由到 Ribbon 消费端。输入 `http://localhost:9999/api-b/feign`，可以看到 Feign 返回的数据。

```
{"show": "hello 张三\nUser{name='李四', age=20}\nUser{name='王五', age=39}\n"}
```

(6) 为了测试过滤器，需要重新启动工程 chapter-05-05，然后再执行上面两个请求，可以在服务器的日志中查看到结果。

```
2017-12-01 10:58:19.155 INFO 10872 --- [nio-9999-exec-4]
c.c.skyme.filter.PreRequestLogFilter : send GET request to
http://localhost:9999/api-b/feign
2017-12-01 10:58:19.380 INFO 10872 --- [nio-9999-exec-6]
c.c.skyme.filter.PreRequestLogFilter : send GET request to
http://localhost:9999/api-b/feign
```

总结

在线上环境，一般建议使用 Zuul 集群进行横向的扩展，提高集群的并发能力，并且将不同的业务根据业务类型进行分组，也就是我们经常提到的冷热分离，根据服务调用次数，完成热服务的拆分。

5.6 消息总线（Spring Cloud Bus）

Spring Cloud Bus 将 Spring 的事件处理机制和消息中间件消息的发送和接收整合起来，可以轻松地将分布式应用中连接有消息中间件的多个服务节点连接起来，实现消息互通。

介绍

Spring Cloud 本身实现了变量修改 `/bus/env` 和 `/bus/refresh` 两个接口，我们需要扩展一个自己的刷新缓存的接口来应对业务需求。业务代码是看源码修改出来的。

Spring Cloud Bus 通过轻量消息代理连接各个分布的节点。Spring Cloud Bus 的一个核心思想是通过分布式的启动器对 Spring Boot 应用进行扩展，也可以用来建立一个多个应用之间的通

信频道。Spring Cloud 本身实现了消息总线机制，机制如图 5-13 所示。

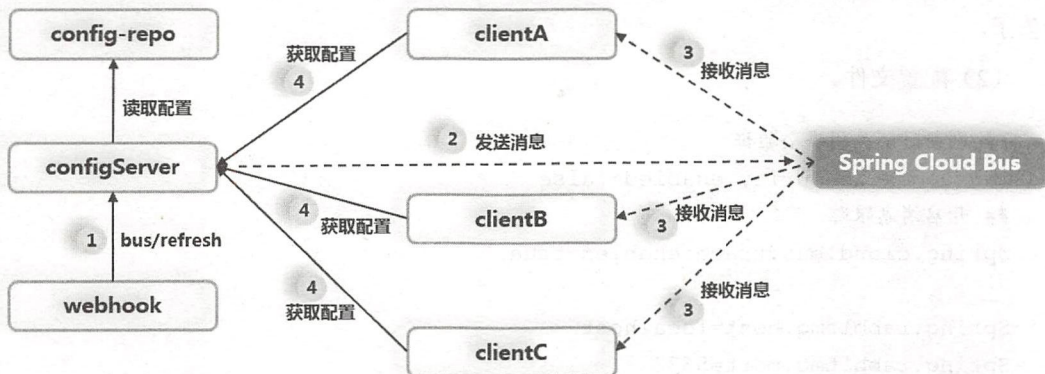


图 5-13 Spring Cloud Bus 消息总线

利用 Spring Cloud Bus 做配置更新的步骤：

- (1) 提交代码触发 post 请求给 bus/refresh。
- (2) Server 端接收到请求并发送给 Spring Cloud Bus。
- (3) Spring Cloud Bus 接到消息并通知给其他客户端。
- (4) 其他客户端接收到通知，请求 Server 端获取最新配置，全部客户端均获取到最新的配置。

Spring Cloud Bus 需要依赖一种 MQ，比较流行的有 RabbitMQ 和 Kafka。

开源地址：<https://github.com/Spring-cloud/Spring-cloud-bus>。

实现

- (1) 添加依赖。

```

<!--消息总线-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>Spring-cloud-starter-bus-amqp</artifactId>
</dependency>
<!--config-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>Spring-cloud-config-monitor</artifactId>
</dependency>
  
```

其中 bus-amqp 就是消息总线，利用 RabbitMQ 来进行消息通信。monnitor 就是在 Server 端设置一个监听，来接收我们发送的 post 信息，告诉它我们更新了配置，可以开始刷新已经注入的值了。

(2) 配置文件。

```
## 刷新时，关闭安全验证
management.security.enabled=false
## 开启消息跟踪
Spring.cloud.bus.trace.enabled=true

Spring.rabbitmq.host=localhost
Spring.rabbitmq.port=5672
Spring.rabbitmq.username=admin
Spring.rabbitmq.password=123456
```

配置文件需要增加 RebbitMQ 的相关配置，可以使用 management.security.enabled 设置属性变化后不使用密码验证。

测试

依次启动 chapter-05-01-eureka-server、chapter-05-04-config-server、Spring-cloud-config-client 项目，分别测试一下服务端和客户端是否正确运行，然后通过客户端模拟请求，当值发生变化的时候通过控制台监控值得变化。

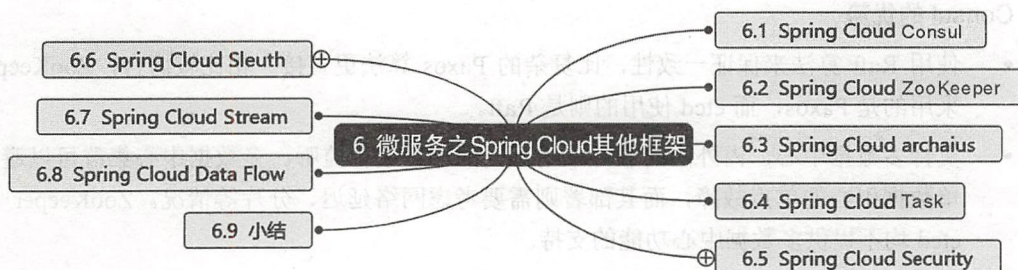
5.7 小结

Spring Cloud 是一套微服务组件，合理地利用这一套组件就能够构建出强大的微服务系统，能够弹性伸缩，动态可扩展。

Spring Cloud 中的 Eureka 能够实现高可用的注册中心，Feign 能够实现服务的调用，并且实现负载均衡。Hystrix 能够有效地防止雪崩，应对高流量的冲击。配置中心能够实现配置信息动态修改，更加灵活地对线上环境进行动态修改。API 网关能够实现安全、动态配置等，并且能够实现高可用，配合 Spring Cloud 的其他组件，可以架构出非常实用的高可用架构。

6 chapter

第 6 章 微服务之 Spring Cloud 其他框架



Spring Cloud 是一套微服务工具集，上一章介绍的是基础组件，本章会扩展 Spring Cloud 的其他工具。同时给出一个样例系统，通过对样例系统的学习，读者即可对微服务的设计架构有一定的了解和认识。

下面我们看一下 Spring Cloud 的其他工具集。

6.1 Spring Cloud Consul

对于注册中心，Consul 是一个新的选择。

基于 Mozilla Public License 2.0 的协议进行开源，Consul 支持健康检查，并允许 HTTP 和 DNS 协议调用 API 存储键值对。一致性协议采用 Raft 算法，用来保证服务的高可用。使用 GOSSIP 协议管理成员和广播消息，并且支持 ACL 访问控制。

Consul 的使用场景

- Docker 实例的注册与配置共享；
- CoreOS 实例的注册与配置共享；
- vitess 集群；
- SaaS 应用的配置共享；
- 与 confd 服务集成，动态生成 Nginx 和 Haproxy 配置文件。

Consul 的优势

- 使用 Raft 算法来保证一致性，比复杂的 Paxos 算法更直接。相比较而言，ZooKeeper 采用的是 Paxos，而 etcd 使用的则是 Raft。
- 支持多数据中心，内外网的服务采用不同的端口进行监听。多数据中心集群可以避免单数据中心的单点故障，而其部署则需要考虑网络延迟、分片等情况。ZooKeeper 和 etcd 均不提供多数据中心功能的支持。
- 支持健康检查，etcd 不提供此功能。
- 支持 HTTP 和 DNS 协议接口。ZooKeeper 的集成较为复杂，而 etcd 只支持 HTTP 协议。
- 官方提供 Web 管理界面，etcd 无此功能。

Consul 的角色

Client: 客户端，无状态，将 HTTP 和 DNS 接口请求转发给局域网内的服务端集群。

Server: 服务端，保存配置信息，高可用集群，在局域网内与本地客户端通信，通过广域网与其他数据中心通信。每个数据中心的 Server 数量推荐为 3 个或 5 个。

CentOS 下安装 Consul:

Linux 64bit:

```
wget https://releases.hashicorp.com/consul/0.8.4/consul_0.8.4_linux_amd64.zip
unzip consul_0.8.4_linux_amd64.zip
sudo mv consul /bin
```

Maven 中需要添加如下配置:

```
<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>Spring-cloud-starter-consul-all</artifactId>
  </dependency>
</dependencies>

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>Spring-cloud-consul-dependencies</artifactId>
      <version>1.3.2.RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

安装完成后, 在 Spring Cloud 中的配置需要改为如下配置:

```
Spring:
  cloud:
    consul:
      host: localhost
      port: 8500
```

@EnableDiscoveryClient 使应用程序成为 Consul “服务”, 所以, 系统启动的时候, 就会向 Consul 注册一个服务, 提供给客户端调用。

6.2 Spring Cloud ZooKeeper

ZooKeeper (ZK) 是一个分布式的, 开放源码的分布式应用程序协调服务, 是 Google 的 Chubby 一个开源的实现, 它是集群的管理者, 监视着集群中各个节点的状态, 根据节点提交的反馈进行下一步合理操作。最终, 将简单易用的接口和性能高效、功能稳定的系统提供给用户。

ZooKeeper 的数据模型很简单, 由一系列被称为 ZNode 的数据节点组成, 与传统的磁盘文件系统不同的是, ZK 将全量数据存储在内存中, 可谓高性能; 而且支持集群, 可谓高可用; 另外支持事件监听。这些特点决定了 ZK 特别适合作为注册中心。

ZooKeeper 的角色包括:

- 领导者 (leader), 负责进行投票的发起和决议, 更新系统状态。
- 学习者 (learner), 包括跟随者 (follower) 和观察者 (observer), follower 用于接收客户端请求并向客户端返回结果, 在选主过程中参与投票。
- Observer 可以接收客户端连接, 将写请求转发给 leader, 但 observer 不参加投票过程, 只同步 leader 的状态, observer 的目的是为了扩展系统, 提高读取速度。
- 客户端 (client), 请求发起方。

ZooKeeper 的核心是原子广播, 这个机制保证了各个 Server 之间的同步。实现这个机制的协议叫作 Zab 协议。Zab 协议有两种模式, 它们分别是恢复模式 (选主) 和广播模式 (同步)。当服务启动或者在领导者崩溃后, Zab 就进入了恢复模式, 当领导者被选举出来, 且大多数 Server 完成了和 leader 的状态同步以后, 恢复模式就结束了。状态同步保证了 leader 和 Server 具有相同的系统状态。

为了保证事务的顺序一致性, ZooKeeper 采用了递增的事务 id 号 (zxid) 来标识事务。所有的提议 (proposal) 都在被提出的时候加上了 zxid。实现中 zxid 是一个 64 位的数字, 它的高 32 位是 epoch, 用来标识 leader 关系是否改变, 每一个 leader 被选出来, 它都会有一个新的 epoch, 标识当前属于那个 leader 的统治时期。低 32 位用于递增计数。

每个 Server 在工作过程中有三种状态。

- LOOKING: 当前 Server 不知道 leader 是谁, 正在搜寻。
- LEADING: 当前 Server 即为选举出来的 leader。
- FOLLOWING: leader 已经选举出来, 当前 Server 与之同步。

在整个服务注册与发现的设计中, 最重要的是如何存储服务的注册信息。ZooKeeper 的数据模型如图 6-1 所示。

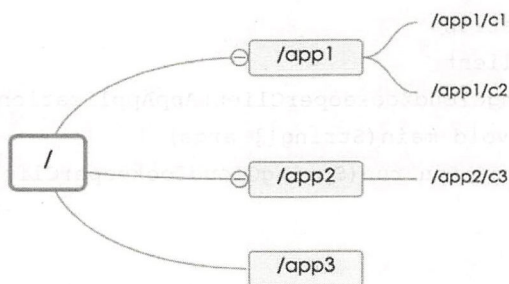


图 6-1 ZooKeeper 数据模型

ZooKeeper 数据模型结构与 UNIX 文件系统很类似，是一个树状层次结构。每个节点叫作 Znode，节点可以拥有子节点，同时允许将少量数据存储在該节点下。客户端可以通过监听节点的数据变更和子节点变更来实时获取 Znode 的变更。

Spring Cloud 同样能够集成 ZooKeeper。ZooKeeper 既可以作为注册中心使用，也可以作为配置中心使用。

安装 ZooKeeper，从官网下载 ZooKeeper 并解压缩，然后给工程添加 Maven 依赖。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>Spring-cloud-starter-zookeeper-discovery</artifactId>
</dependency>
```

application.properties 的配置信息如下：

```
## 配置应用名称
Spring.application.name=Spring-cloud-zookeeper-client-app

## 配置服务端口
server.port=8080

## 关闭安全控制
management.security.enabled=false

## 配置 ZooKeeper 的地址
Spring.cloud.zookeeper.connect-string=localhost:2181
```

作为服务中心的代码如下：

```

@SpringBootApplication
@EnableDiscoveryClient
public class SpringCloudZookeeperClientAppApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringCloudZookeeperClientAppApplication.
class, args);
    }
}

```

获取注册服务列表的代码如下：

```

public List<String> serviceUrl() {
    List<ServiceInstance> list = discoveryClient.getInstances(instanceName);
    List<String> services = new ArrayList<>();
    if (list != null && list.size() > 0 ) {
        list.forEach(serviceInstance -> {
            services.add(serviceInstance.getUri().toString());
        });
    }
    return services;
}

```

6.3 Spring Cloud archaius

archaius 是 Netflix 公司开源项目之一，基于 Java 的配置管理类库，主要用于多配置存储的动态获取。主要功能是对 Apache common configuration 类库的扩展。在云平台开发中可以将其用作分布式配置管理依赖构件。

使用 Maven 方式引入：

```

<dependency>
    <groupId>com.netflix.archaius</groupId>
    <artifactId>archaius-core</artifactId>
    <version>0.6.0</version>
</dependency>

```

使用本地配置文件作为配置源。

默认的，archaius 将查找 classpath 下名为 config.properties 的文件并读取，这个配置文件可

以包含在一个 jar 包的根路径下。

另外，可以使用属性 `archaius.configurationSource.additionalUrls` 来包含 URL 形式的文件，多个文件用逗号分隔。

可以使用下面的 API 在程序中得到你需要的属性：

```
DynamicIntProperty prop =
    DynamicPropertyFactory.getInstance().getIntProperty("myProperty",
DEFAULT_VALUE);
// prop.get() may change value at runtime
myMethod(prop.get());
```

默认的，archaius 会每隔一分钟去重新加载一次属性配置。

可以修改的系统的系统属性如表 6-1 所示。

表 6-1 archaius 修改的系统属性

Operation	HTTP action	Notes
<code>archaius.configurationSource.defaultFileName</code>	指定 Archaius 默认加载的配置源属性文件名，默认为 <code>classpath:config.properties</code>	<code>config.properties</code>
<code>archaius.fixedDelayPollingScheduler.initialDelayMills</code>	延迟加载，默认为 30 秒	30000
<code>archaius.fixedDelayPollingScheduler.delayMills</code>	两次属性读取时间间隔，默认为 1 分钟	60000

6.4 Spring Cloud Task

Spring Cloud Task 允许用户使用 Spring Cloud 开发和运行短期微服务器，并在云端运行，甚至在 Spring Cloud Data Flow 中运行。只需添加 `@EnableTask` 并运行应用程序作为 Spring Boot 应用程序即可。

添加依赖：

```
<dependencies>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>Spring-cloud-starter-task</artifactId>
<version>1.2.2.RELEASE</version>
</dependency>
</dependencies>
```

只要 Spring Cloud Task 在类路径中, 任何 Spring 引导应用程序 `@EnableTask` 将记录引导应用程序的开始和结束, 以及配置的任务存储库中的任何未捕获的异常。

示例代码如下:

```
@SpringBootApplication
@EnableTask
public class ExampleApplication {
    @Bean
    public CommandLineRunner commandLineRunner() {
        return strings ->
            System.out.println("Executed at :" +
                new SimpleDateFormat().format(new Date()));
    }
    public static void main(String[] args) {
        SpringApplication.run(ExampleApplication.class, args);
    }
}
```

6.5 Spring Cloud Security

微服务的安全需要从以下几个角度入手。

➤ 保护开发生命周期

我们需要向服务中引入变更, 加以测试, 而后立即将成果部署至生产环境。为了确保在代码层面中不存在安全漏洞, 我们需要制定规划以进行静态代码分析与动态测试。

➤ DevOps 安全

微服务部署模式可谓多种多样, 但其中使用最为广泛的当数主机服务模式。其中, 主机指定的并不一定是物理设备, 也很可能属于容器(Docker)。我们需要对容器层面的安全进行关注。

➤ 应用级别安全

即如何验证用户身份并对其微服务访问操作进行控制, 以及保障不同微服务之间的通信安全。

首先解释两个长得很像、容易混淆的单词, Authentication (鉴定、认证) 和 Authorization (授权)。

Authentication 就是要证明你是谁。举个例子, 你告诉别人你的名字叫 Alice, 怎样让别人确信你就是 Alice, 这就是 Authentication。

Authorization 则是当别人已经相信是你以后，你是不是被允许做某件事。比如，当你已经证明你就是 Alice 了，你可以查你自己的信用卡刷卡记录，但不能查 Bob 的刷卡记录，这就是 Authorization。

下面我们一起看一下微服务中常用的安全规范。

6.5.1 HTTP Basic Authentication

在访问一个需要 HTTP Basic Authentication 的 URL 时，如果没有提供用户名和密码，则服务器就会返回 401，如果直接在浏览器中打开，则浏览器会提示你输入用户名和密码。

HTTP Basic 指的就是最简单的 Authentication 协议。简单到什么程度呢？直接告诉服务器你的用户名(username)和密码(password)。这里假设我们的用户名是 zhangfeng，密码是 123456。

我们使用 curl 访问服务器：

```
curl -u zhangfeng:123456 http://kiwiserver.com/secret -v
```

request 头部：

```
GET /secret HTTP/1.1
Authorization: Basic emhhbmdmZW5nIDEyMzQ1Ng==
...
```

我们这里看到发送的 request 头部中含有 Authentication 字段，其值为 Basic emhhbmdmZW5nIDEyMzQ1Ng==。Basic 表示使用的是 HTTP Basic Authentication。而 emhhbmdmZW5nIDEyMzQ1Ng==则是由“Alice:123456”进行 Base64 编码以后得到的结果。

response 头部：

```
HTTP/1.1 200 OK
...
```

因为我们输入的是正确的用户名和密码，所以服务器会返回 200，表示验证成功。如果我们用错误的用户名和密码来发送请求，则会得到类似如下含有 401 错误的 response 头部：

```
HTTP/1.1 401 Bad credentials
WWW-Authenticate: Basic realm="Spring Security Application"
...
```

乍看起来好像 HTTP Basic 还挺不错的，emhhbmdmZW5nIDEyMzQ1Ng==已经让人很难看

出来用户名密码是什么了。但是,要知道 Base64 编码是可逆的。也就是说,我们可以通过 decode Base64 的编码还原用户名和密码。

在命令行输入如下命令:

```
echo emhhbmdmZW5nIDEyMzQ1Ng== | base64 -D
```

得到:

```
zhangfeng:123456
```

轻松解密。试想,如果一个人通过一定的方法截获了 zhangfeng 向服务器发送的请求,那岂不是很容易就能够得到用户名和密码了吗?所以,为了保证用户的安全,我们不会直接通过 HTTP 的方式使用 Basic Authentication,而是会使用 HTTPS,这样更安全一些。

这种安全验证方式级别比较低,很容易受到 Replay Attack 攻击。

6.5.2 JWT

JSON Web Token (JWT) 是为了在网络应用环境间传递声明而执行的一种基于 JSON 的开放标准(RFC 7519)。该 Token 被设计为紧凑且安全的,特别适用于分布式站点的单点登录(SSO)场景。JWT 的声明一般被用来在身份提供者和服务提供者间传递被认证的用户身份信息,以便从资源服务器获取资源,也可以增加一些额外的其他业务逻辑所必需的声明信息,该 Token 也可直接被用于认证,还可被加密。

已签名 JWT 被称为 JWS (即 JSON Web 签名),而加密 JWT 则被称为 JWE (即 JSON Web 加密)。事实上,JWT 并不会以自身原始方式存在,要么作为 JWS,要么作为 JWE,像是一种抽象类,JWS 与 JWE 为其具体实现方式。

JWT 的构成

第一部分我们称它为头部(header),第二部分我们称为载荷(payload,类似于飞机上承载的物品),第三部分是签证(signature)。

➤ header

JWT 的头部承载两部分信息:

- 声明类型,这里是 JWT。
- 声明加密的算法,通常直接使用 HMAC SHA256。

完整的头部就像下面这样的 JSON:

```
{
  'typ': 'JWT',
  'alg': 'HS256'
}
```

然后将头部进行 base64 加密（该加密是可以对称解密的），构成了第一部分。

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9
```

➤ payload

载荷就是存放有效信息的地方。这个名字像是特指飞机上承载的货品，这些有效信息包含三个部分：

- 标准中注册的声明；
- 公共的声明；
- 私有的声明。

标准中注册的声明，建议但不强制使用。

- iss: JWT 签发者；
- sub: JWT 所面向的用户；
- aud: 接收 JWT 的一方；
- exp: JWT 的过期时间，这个过期时间必须大于签发时间；
- nbf: 定义在什么时间之前，该 JWT 都是不可用的；
- iat: JWT 的签发时间；
- jti: JWT 的唯一身份标识，主要用来作为一次性 Token，从而回避重放攻击。

公共的声明：

公共的声明可以添加任何信息，一般添加用户的相关信息或其他业务需要的必要信息，但不建议添加敏感信息，因为该部分在客户端可解密。

私有的声明：

私有的声明是提供者和消费者所共同定义的声明，一般不建议存放敏感信息，因为 base64 是对称解密的，意味着该部分信息可以归类为明文信息。

定义一个 payload:

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

然后将其进行 base64 加密, 得到 JWT 的第二部分。

```
eyJzdWIiOiIxMjMONTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWYWRtaW4iOnRydWV9
```

➤ signature

JWT 的第三部分是一个签证信息, 这个签证信息由三部分组成:

- header (base64 后的);
- payload (base64 后的);
- secret。

这部分需要 base64 加密后的 header 和 base64 加密后的 payload 使用 “.” 连接组成的字符串, 通过 header 中声明的加密方式进行加盐 secret 组合加密, 然后就构成了 JWT 的第三部分。

```
// javascript
var encodedString = base64UrlEncode(header) + '.' +
base64UrlEncode(payload);
var signature = HMACSHA256(encodedString, 'secret'); //
TJVA95OrM7E2cBab30RMhrHDcEfxjoYZgeFONFh7HgQ
```

将这三部分用 “.” 连接成一个完整的字符串, 构成了最终的 JWT:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjMONTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RMhrHDcEfxjoYZgeFONFh7HgQ
```

注意: secret 是保存在服务器端的, JWT 的签发生成也是在服务器端的, secret 就是用来进行 JWT 的签发和 JWT 的验证, 所以, 它就是服务端的私钥, 在任何场景都不能泄露出去。一旦客户端得知这个 secret, 那就意味着客户端可以自我签发 JWT 了。

JWT 流程如图 6-2 所示。

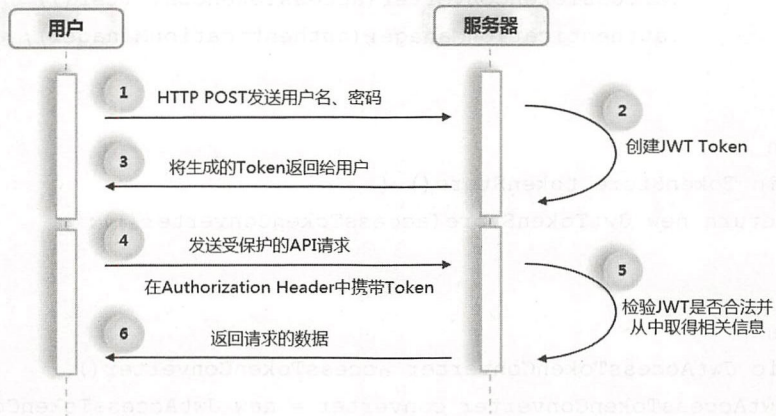


图 6-2 JWT 流程

- 用户发起 HTTP POST 请求，并且携带用户名和密码。
- 服务器通过验证后生成 Token。
- 服务器将生成的 Token 返回给客户端。
- 客户端发送受保护的 API 请求，在 Authorization Header 中携带 Token。
- 服务端接收到请求后校验 JWT 是否合法并从中获得相关信息。
- 认证通过调用相应的方法返回请求的数据。

在工程中使用，首先添加 spring-security-jwt 的 Maven 依赖，代码如下：

```
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-jwt</artifactId>
</dependency>
```

然后编写授权验证服务，代码如下：

```
@Configuration
@EnableAuthorizationServer
public class OAuth2AuthorizationServerConfig extends AuthorizationServer-
ConfigurerAdapter {
  @Override
  public void configure(AuthorizationServerEndpointsConfigurer endpoints)
throws Exception {
    endpoints.tokenStore(tokenStore())
  }
}
```

```

        .accessTokenConverter(accessTokenConverter())
        .authenticationManager(authenticationManager);
    }

    @Bean
    public TokenStore tokenStore() {
        return new JwtTokenStore(accessTokenConverter());
    }

    @Bean
    public JwtAccessTokenConverter accessTokenConverter() {
        JwtAccessTokenConverter converter = new JwtAccessTokenConverter();
        converter.setSigningKey("admin");
        return converter;
    }

    @Bean
    @Primary
    public DefaultTokenServices tokenServices() {
        DefaultTokenServices defaultTokenServices = new DefaultTokenServices();
        defaultTokenServices.setTokenStore(tokenStore());
        defaultTokenServices.setSupportRefreshToken(true);
        return defaultTokenServices;
    }
}

```

资源服务的配置代码如下：

```

@Configuration
@EnableResourceServer
public class OAuth2ResourceServerConfig extends ResourceServerConfigurer-
Adapter {
    @Override
    public void configure(ResourceServerSecurityConfigurer config) {
        config.tokenServices(tokenServices());
    }

    @Bean

```



```

public TokenStore tokenStore() {
    return new JwtTokenStore(accessTokenConverter());
}

@Bean
public JwtAccessTokenConverter accessTokenConverter() {
    JwtAccessTokenConverter converter = new JwtAccessTokenConverter();
    converter.setSigningKey("123");
    return converter;
}

@Bean
@Primary
public DefaultTokenServices tokenServices() {
    DefaultTokenServices defaultTokenServices = new DefaultTokenServices();
    defaultTokenServices.setTokenStore(tokenStore());
    return defaultTokenServices;
}
}

```

客户端使用，在使用时我们可以添加一些额外的字段，客户端代码如下：

```

public class CustomTokenEnhancer implements TokenEnhancer {
    @Override
    public OAuth2AccessToken enhance(
        OAuth2AccessToken accessToken,
        OAuth2Authentication authentication) {
        Map<String, Object> additionalInfo = new HashMap<>();
        additionalInfo.put("organization", authentication.getName() +
            randomAlphabetic(4));
        ((DefaultOAuth2AccessToken) accessToken).setAdditionalInformation(
            additionalInfo);
        return accessToken;
    }
}

```

然后在验证服务上添加配置，代码如下：

```

@Override
public void configure(AuthorizationServerEndpointsConfigurer endpoints)
throws Exception {
    TokenEnhancerChain tokenEnhancerChain = new TokenEnhancerChain();
    tokenEnhancerChain.setTokenEnhancers(
        Arrays.asList(tokenEnhancer(), accessTokenConverter());

    endpoints.tokenStore(tokenStore())
        .tokenEnhancer(tokenEnhancerChain)
        .authenticationManager(authenticationManager);
}

@Bean
public TokenEnhancer tokenEnhancer() {
    return new CustomTokenEnhancer();
}

```

运行后，可以得到如下的 JSON 信息：

```

{
  "user_name": "john",
  "scope": [
    "foo",
    "read",
    "write"
  ],
  "organization": "johnIiCh",
  "exp": 1458126622,
  "authorities": [
    "ROLE_USER"
  ],
  "jti": "e0ad1ef3-a8a5-4eef-998d-00b26bc2c53f",
  "client_id": "fooClientIdPassword"
}

```

也可以在 JavaScript 中使用，代码如下：

```
<p class="navbar-text navbar-right">{{organization}}</p>
```



```

<script type="text/javascript"
  src="https://cdn.rawgit.com/auth0/angular-jwt/master/dist/angular-
jwt.js">
</script>

<script>
  var app = angular.module('myApp', ["ngResource", "ngRoute", "ngCookies",
"angular-jwt"]);

  app.controller('mainCtrl', function($scope, $cookies, jwtHelper,...) {
    $scope.organization = "";

    function getOrganization(){
      var token = $cookies.get("access_token");
      var payload = jwtHelper.decodeToken(token);
      $scope.organization = payload.organization;
    }
    ...
  });

```

6.5.3 OAuth 2

OAuth 是一个关于授权（authorization）的开放网络标准，在全世界得到广泛应用，目前的版本是 2.0 版。

OAuth 的几个专有名词如下。

- Third-party application: 第三方应用程序。
- HTTP service: HTTP 服务提供商。
- Resource Owner: 资源所有者。
- User Agent: 用户代理。
- Authorization Server: 认证服务器，即服务提供商专门用来处理认证的服务器。
- Resource Server: 资源服务器，即服务提供商存放用户生成的资源的服务器。它与认证服务器可以是同一台服务器，也可以是不同的服务器。

OAuth 在“客户端”与“服务提供商”之间设置了一个授权层（authorization layer）。“客户

端”不能直接登录“服务提供商”，只能登录授权层，以此将用户与客户端区分开来。“客户端”登录授权层所用的令牌（Token）与用户的密码不同。用户可以在登录时指定授权层令牌的权限范围和有效期。

OAuth 2.0 定义了四种授权方式：

- 授权码模式（authorization code）；
- 简化模式（implicit）；
- 密码模式（resource owner password credentials）；
- 客户端模式（client credentials）。

OAuth 2.0 的核心是取得 access_token，四种授权模式殊途同归，最终都是要取得 access_token。

OAuth 2.0 的基本流程

用户通过浏览器访问一个应用，比如我要上某网站学习。

- 网站要求我登录，我选择使用 QQ 登录，这里的 QQ 登录就是那个认证服务器。
- 这时网站提供的 QQ 登录链接会把我带到 QQ 登录页面。
- 在 QQ 的登录页面完成登录后，选择授权，也就是允许网站获取我的资料。
- 这个时候我们看到浏览器经过几次跳转后返回要学习的网站，这时我们已经完成了登录。

上面就是一个简单的 OAuth 2.0 的登录流程。

6.5.4 Spring Cloud Security

Spring Cloud Security 基于 OAuth 2 这个开放网络的安全标准，提供了微服务环境下的单点登录、资源授权、令牌管理等功能，提供了一套用于构建安全的原语级应用程序和最小化服务。

功能

- 在 Zuul 代理中将 SSO 令牌从前端转发到后端服务；
- 资源服务器之间的中继令牌；
- 一个拦截器可以使一个 Feign 客户端的行为像 OAuth2RestTemplate（获取令牌等）；
- 在 Zuul 代理中配置下游认证。

使用

在 Maven 中添加依赖:

```

<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>Spring-cloud-security</artifactId>
    <version>1.1.4.BUILD-SNAPSHOT</version>
  </dependency>
</dependencies>
<repositories>
  <repository>
    <id>Spring-snapshots</id>
    <name>Spring Snapshots</name>
    <url>https://repo.spring.io/libs-snapshot</url>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
  </repository>
</repositories>

```

如果使用 Zuul, 则可以要求它将 OAuth 2 访问令牌转发到其代理的服务器, 只需添加如下代码:

```

@SpringBootApplication
@EnableOAuth2Sso
@EnableZuulProxy
class Application {

}

```

这样就实现了安全代理认证。

6.6 Spring Cloud Sleuth

Spring Cloud Sleuth 提供了一套完整的服务跟踪的解决方案。

跟踪原理: pom 中依赖 Spring-cloud-starter-sleuth 包后, 每次链路请求都会添加一串追踪信息, 格式是[server-name, main-traceId,sub-spanId,boolean]。

- 第一个参数：服务节点名称；
- 第二个参数：一条链路唯一的 ID，叫作 TraceID；
- 第三个参数：链路中每一环的 ID，叫作 SpanID；
- 第四个参数：是否将信息输出到 Zipkin 等服务收集和展示。

Spring Cloud Sleuth 要解决的问题：

(1) 服务追踪，一条链路上的所有处理分配统一的 TraceID，通过这个唯一标识就可以找到完整的处理链路。

(2) 每一环的微服务节点处理时我们再分配一个独立的 SpanID，这样请求何时到达节点、何时离开节点都有追踪的依据，由此就可以判断出每一跳的延时情况。

6.6.1 服务端

首先，添加 Maven 依赖配置，代码如下：

```
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter</artifactId>
</dependency>
<dependency>
<groupId>io.zipkin.java</groupId>
<artifactId>zipkin-server</artifactId>
</dependency>
<dependency>
<groupId>io.zipkin.java</groupId>
<artifactId>zipkin-autoconfigure-ui</artifactId>
<scope>runtime</scope>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope>
</dependency>
```

然后写一个启动类 ZipkinServer，代码如下：

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
```



```
import zipkin.server.EnableZipkinServer;

@EnableZipkinServer
@SpringBootApplication
public class ZipkinServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(ZipkinServerApplication.class, args);
    }
}
```

接着编写配置文件 application.yml，代码如下：

```
server:
  port: 11008
spring:
  application:
    name : microservice-zipkin-server
```

启动应用。

6.6.2 客户端

添加依赖，引入 Zipkin 客户端自动配置相关依赖，代码如下：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-sleuth-zipkin</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
```

然后在客户端的配置文件中指向服务端的地址，代码如下：

```
spring:
  application:
    name: microservice-zipkin-client-backend
```

```
zipkin:
  base-url: http://localhost:11008
```

启动客户端应用并访问相关的服务接口，我们就能够在服务端的界面上看到请求的数据，效果如图 6-3 所示。

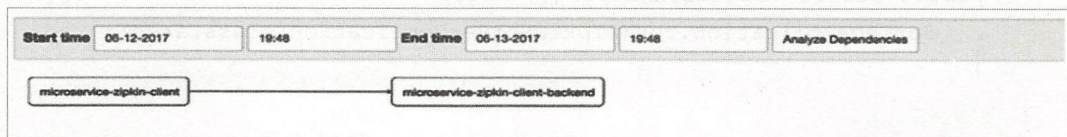


图 6-3 Spring Cloud Sleuth

在测试的过程中我们会发现，有时程序刚刚启动后，刷新几次，并不能看到任何数据，原因就在于 spring-cloud-sleuth 收集信息是有一定的比率的，默认的采样率是 0.1。配置此值的方式是在配置文件中增加 spring.sleuth.sampler.percentage 参数配置（如果不配置默认 0.1），如果我们调大此值为 1，可以看到信息收集会更及时。但是当这样调整后，会发现 REST 接口调用速度比 0.1 的情况下慢了很多。即使在 0.1 的采样率下，多次刷新 consumer 的接口，也会发现对同一个请求两次耗时信息相差非常大。如果取消 spring-cloud-sleuth 后再测试，会发现并没有这种情况，可以看到这种方式追踪服务调用链路会给业务程序性能带来一定的影响。

6.7 Spring Cloud Stream

Spring Cloud Stream 对分布式消息的各种需求进行了抽象，包括发布订阅、分组消费、消息分片等功能，实现了微服务之间的异步通信。Spring Cloud Stream 也集成了第三方的 RabbitMQ 和 Apache Kafka 作为消息队列的实现。而 Spring Cloud Bus 基于 Spring Cloud Stream，主要提供了服务间的事件通信（比如刷新配置）。Spring Cloud Stream 是一个构建消息驱动微服务的框架。

Spring Cloud Stream 的架构如图 6-4 所示。

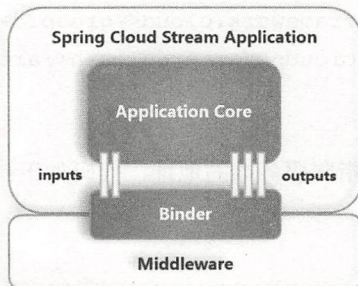


图 6-4 Spring Cloud Stream 架构

应用程序通过 inputs 或者 outputs 来与 Spring Cloud Stream 中的 Binder 交互, 通道通过指定中间件的 Binder 实现与外部代理连接。而 Spring Cloud Stream 的 Binder 负责与中间件交互。业务开发者不再关注具体消息中间件, 只需关注 Binder 对应用程序提供的抽象概念来使用消息中间件实现业务即可。

➤ Binder

Binder 是 Spring Cloud Stream 的一个抽象概念, 是应用与消息中间件之间的黏合剂。目前 Spring Cloud Stream 实现了 Kafka 和 RabbitMQ 的 Binder。

➤ Publish-Subscribe

消息的发布 (Publish) 和订阅 (Subscribe) 是事件驱动的经典模式。Spring Cloud Stream 的数据交互也基于这个思想。生产者把消息通过某个 topic 广播出去 (Spring Cloud Stream 中的 destinations)。其他的微服务通过订阅特定 topic 来获取广播出来的消息以触发业务。

➤ Consumer Groups

微服务中动态地缩放同一个应用的数量以此来达到更高的处理能力是非常必要的。对于这种情况, 为防止同一个事件被重复消费, 只需把这些应用放置于同一个 “group” 中, 就能够保证消息只会被其中一个应用消费一次。

➤ Durability

消息事件的持久化是必不可少的。Spring Cloud Stream 可以动态地选择一个消息队列是持久化还是 present。

➤ Bindings

Bindings 通过配置把应用和 Spring Cloud stream 的 Binder 绑定在一起, 之后只需修改 Binding 的配置来实现动态修改 topic、exchange、type 等一系列信息即可, 而不需要修改一行代码。

那么使用 Spring Cloud Stream 的最大好处是什么呢?

Spring Cloud Stream 最大的方便之处, 莫过于抽象了事件驱动的一些概念, 对于消息中间件的进一步封装, 可以做到代码层面对中间件的无感知, 甚至动态地切换中间件、切换 topic。使得微服务的开发高度解耦, 服务可以关注更多自己的业务流程。

快速开始

添加 Maven 依赖:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
```

```

<artifactId>Spring-cloud-stream-dependencies</artifactId>
<version>Brooklyn.SR3</version>
<type>pom</type>
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>
<dependencies>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>Spring-cloud-stream</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>Spring-cloud-starter-stream-kafka</artifactId>
</dependency>
</dependencies>

```

只要 Spring Cloud Stream 和 Spring Cloud Stream Binder 在类路径中,任何具有@EnableBinding 的 Spring Boot 应用程序都将绑定到总线提供的外部代理。

以下是接收外部消息的简单接收器应用程序:

```

@SpringBootApplication
@EnableBinding(Sink.class)
public class VoteRecordingSinkApplication {

    public static void main(String[] args) {
        SpringApplication.run(VoteRecordingSinkApplication.class, args);
    }

    @StreamListener(Sink.INPUT)
    public void processVote(Vote vote) {
        votingService.recordVote(vote);
    }
}

```

@EnableBinding 注释需要一个或多个接口作为参数(在这种情况下,该参数是单个 Sink

接口)。接口声明输入和/或输出通道。Spring Cloud Stream 提供了接口 Source、Sink 和 Processor，还可以定义自己的界面。

Spring Cloud Stream 将创建一个界面的实现，可以在应用程序中通过自动连接来使用它，如下面的测试用例示例。

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = VoteRecordingSinkApplication.class)
@WebAppConfiguration
@DirtiesContext
public class StreamApplicationTests {

    @Autowired
    private Sink sink;

    @Test
    public void contextLoads() {
        assertNotNull(this.sink.input());
    }
}
```

6.8 Spring Cloud Data Flow

数据处理的类型主要集中在三个方面。

- 批处理：多用于处理大量具有可扩展性与分布性的静态数据。
- 实时处理：主要用于处理流式数据（连续不断的无限数据流），这类数据具有分布性与高速率特质。
- 混合计算模式：这种模式是批处理与实时处理的结合，可处理大容量的高速率数据。

Spring Cloud Data Flow 是一个混合计算模型，结合了流数据与批量数据的处理方式。开发者可以通过 Spring Cloud Data Flow，在诸如数据获取、实时分析、批处理等常见用例中执行数据流的创建与编排。Spring Cloud Data Flow 的目标就是为了方便数据工程师，让他们能专注于分析工作和具体的问题。

Spring Cloud Data Flow 是 Spring XD 的修订版，在功能的构成方式上，以及如何协助原生云架构扩展应用方面，都做出了根本性的改变。

Spring Cloud Data Flow 不再使用传统基于组件的架构，而是采用了信息驱动的微服务架构，

这种架构更适合原生云应用平台的原生应用。

Source、job、sink 和 processor 模块都是 Spring Boot 的微服务，可以部署在 Cloud Foundry、Lattice 或 Yarn 集群上。使用这些微服务来部署原生云平台，我们就能创建数据流，并将其输入基于 Yarn、Lattice 或 Cloud Foundry 的目标。特定平台的 SPI（服务提供商接口）可用于基于平台部署的微服务绑定、探索和绑定通道。

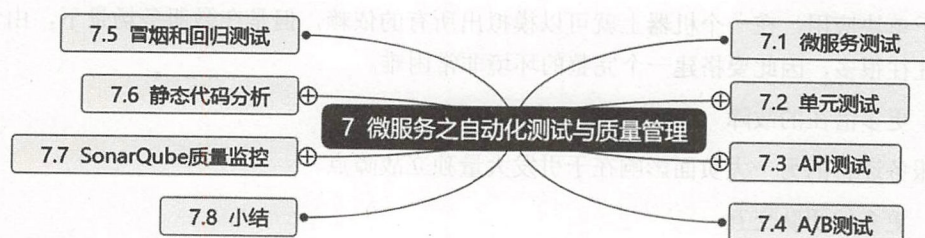
6.9 小结

Spring Cloud 除了常用的一些功能，还包含很多扩展的功能组件，这些组件可以单独加到项目中，也可以和其他 Spring Cloud 组件一起使用，扩展成更强大的组件。

7 chapter

第 7 章

微服务之自动化测试与质量管理



微服务应用程序的另一个好处是更快且更容易更新。当开发者对一个传统的单体应用程序进行变更时，他们必须做详细的 QA 测试，以确保变更不会影响到其他特性或功能。但有了微服务，开发者可以更新应用程序的单个组件。

微服务在带来好处的同时，同样也带来了挑战，具体可以总结为以下几点。

➤ 系统依赖性增加

由单体应用过渡到微服务架构时，需要加入更多的隔离组件，这些组件的加入势必会带来更多的依赖性。也就是说，复杂性会变得更复杂。这同样给测试带来了相应的复杂度，原本只需要从接口层开始的测试，一下子从深度上多出来更多的依赖，测试的工作量也是成倍地增加。

我们拿一个单体应用有 10 个接口的工程举例，原本测试只需要测试这 10 个接口就可以，但是使用微服务构建之后，接口的数量明显增加，每一个接口所衍生出来的新接口都需要测试，这样可能是一个倍数的关系。所以，微服务的接口抽象及经验就显得非常重要。

➤ 并行开发

接口数量的增加还会带来一个问题，就是原本一个团队维护的项目，被拆分之后，可能是两个或者更多的团队来进行维护。在需求变化时，一个接口的变动，导致的可能是连锁反应，当其中的一个接口出现延期时，整个测试计划就很难得到保证。团队需要等待其他团队以完整相关微服务的并行开发，微服务数量越多，需要考虑的对象就越广泛，这意味着以并行方式开发及发布新功能就变得更加困难。所以，微服务的并行开发同样带来了管理上的挑战。

➤ 与传统测试的冲突

对于单块应用，在一个机器上就可以模拟出所有的依赖，但是在微服务场景下，由于依赖的服务往往很多，因此要搭建一个完整的环境非常困难。

➤ 更多潜在的故障

微服务迁移的另一大负面影响在于引发大量独立故障点。

➤ 更多的团队交互

微服务的方式对于测试来说，意味着需要交互和沟通的人数必然增加，因为一般的团队很难做到需求和实现的高效沟通。沟通成本的增加有可能导致某个重要的功能点没得到优先测试，而不重要的功能点却被优先测试，从而导致整体工期的拖延。

7.1 微服务测试

在微服务中，测试是一个非常重要的环节，相比于常见的三层测试金字塔，在微服务场景下，这个层次可以被扩展为 5 层，如图 7-1 所示。

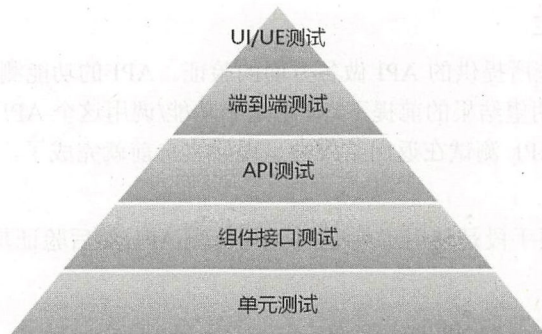


图 7-1 测试金字塔

和测试金字塔的基本原则相同：

- 越往上，越接近业务/最终用户；越往下，越接近开发。
- 越往上，测试用例越少。
- 越往上，测试成本越高（越耗时，失败时的信息越模糊，越难跟踪）。
- 越往上，测试的实现和维护成本就越高，测试速度也越慢。

接下来看一下上面的测试模型。

➤ UI/UE 测试

需要测试的方面很多，包括功能测试、兼容性测试、安装卸载测试等。一般是根据业务模型编写测试用例，然后根据测试用例来判断结果的正确性与否。当然也包括测试用户界面的风格是否满足客户要求，文字是否正确，页面美工是否好看，文字、图片组合是否完美，背景是否美观，操作是否友好，等等。UI 测试还可确保 UI 中的对象按照预期的方式运行，并符合公司或行业的标准，包括用户友好性、人性化、易操作性测试。UI 测试比较主观，与测试人员的喜好有关，比如页面基调颜色刺眼、用户登录页面比较难以找到、文字中出现错别字、页面图片范围太广等都属于 UI 测试中的缺陷。严格来说，所有的测试都包括 UI/UE 测试，不仅仅是在微服务测试中。

➤ 端对端测试

除了测试服务，测试者还需要确保无论使用何种架构构建，应用都必须实现业务目标，同时我们还要测试集成系统运行的完整性。因此在微服务测试方案中，端对端测试占据了重要的角色。除此之外，考虑到在微服务架构中有一些执行相同行为的可移动部件，端对端测试时需要找出覆盖缺口，并确保在架构重构时业务功能不会受到影响。

➤ API 测试

API（Application Programming Interface，简称 API）又称为应用编程接口，就是软件系统

不同组成部分衔接的约定。

API 测试是针对系统所提供的 API 做各方面的验证。API 的功能测试类似于 UI 功能测试，都是在已知输入内容和期望结果的前提下，使用这个功能/调用这个 API 并且验证是否能返回期望的结果。不同的是，API 测试在返回结果被呈现给客户前就完成了，从而对测试环境的依赖会比较小。

API 功能测试的主要手段是使用工具/软件调用待测 API，然后验证是否返回期望的 output。这个 output 通常可能是：

- 返回成功或者失败的状态；
- 一段数据或者 information；
- 跳转到其他 API。

➤ 组件接口测试

尽管单独测试模块非常重要，但测试各个模块能否正确交互，并测试其作为子系统的交互性以查看接口的缺陷同样重要，这项工作可以通过集成测试来完成。集成测试的目的在于：通过集成模块检查路径畅通与否，来确认模块与外部组件的交互情况。执行“网关集成测试”与“持续集成测试”能确保在找到外部组件间的逻辑回归与断裂之处时迅速获得反馈，从而有助于评估各个单独模块中所含逻辑的正确性。

➤ 单元测试

单元测试的范围局限在服务内部，它是围绕着一组相关联的案例编写的。由于单元测试的数量较多，理论上应当是以自动化方式执行的。在微服务中执行单元测试时，必须将协作型单元测试（Sociable Unit Testing）与孤立型单元测试（Solitary Unit Testing）相结合，通过观察其状态变化来检查模块行为，并查看对象及其依赖项之间的交互情况。然而，测试者需要确保在单元测试中，当单元“行为”受限时，“实现”不会受到测试的限制——可以通过不断将单元测试的价值与维护成本/实现受限的成本相对比来做到这一点。

7.2 单元测试

7.2.1 单元测试及覆盖率评估

单元测试就是编写测试代码，用来检测特定的、明确的、细颗粒的功能。单元测试不仅仅用来保证当前代码的正确性，更重要的是用来保证代码修复、改进或重构之后的正确性。

一个良好的单元测试包括三个步骤：

- 准备测试环境和数据；

- 执行目标方法；
- 验证执行结果（判断程序的运行结果是否如你所想）。

在做单元测试时，代码覆盖率常常被拿来作为衡量测试好坏的指标，甚至用代码覆盖率来考核测试任务完成情况，比如代码覆盖率必须达到 80% 或 90%。

那么一般的代码覆盖率都包含哪些呢？通常的评估指标如下：

- 行覆盖率；
- 分支覆盖率；
- 路径覆盖率；
- 条件覆盖率；
- 状态机覆盖率。

目前很多公司已经意识到了单元测试的重要性，但国内坚持写单元测试的团队并不多，其中一个难点在于没有考量，没有很好地执行单元测试覆盖率检测。

7.2.2 JUnit

测试常常是程序员十分厌倦的一个活动。测试能给我们带来什么？了解这些是非常重要的，测试不可能保证一个程序是完全正确的，但是测试却可以增强我们对程序完整的信心，测试可以让我们相信程序做了我们期望它做的事情，测试能够使我们尽早地发现程序的 Bug 和不足。

单元测试（Unit Testing）是指对软件中的最小可测试单元进行检查和验证。比如我们可以测试一个类，或者一个类中的一个方法。单元测试是一个方法层面上的测试，也是最细粒度的测试。用于测试一个类的每一个方法都已经满足了方法的功能要求。JUnit 是 Java 单元测试框架。

JUnit 的基本注释如下。

➤ @BeforeClass

在所有测试方法前执行一次，一般在其中写上整体初始化的代码。

➤ @AfterClass

在所有测试方法后执行一次，一般在其中写上销毁和释放资源的代码。

➤ @Before

在每个测试方法前执行，一般用来初始化方法（比如我们在测试别的方法时，类中与其他测试方法共享的值已经被改变，为了保证测试结果的有效性，我们会在 @Before 注解的方法中重置数据）。

➤ @After

在每个测试方法后执行，在方法执行完成后要做的事情。

➤ @Test(timeout = 1000)

测试方法执行超过 1000 毫秒后算超时，测试将失败。

➤ @Test(expected = Exception.class)

测试方法期望得到的异常类，如果方法执行没有抛出指定的异常，则测试失败。

➤ @Ignore("not ready yet")

执行测试时将忽略掉此方法，如果用于修饰类，则忽略整个类。

➤ @RunWith

在 JUnit 中有很多个 Runner，它们负责调用测试代码，每一个 Runner 都有各自的特殊功能，要根据需要选择不同的 Runner 来运行测试代码。如果只是简单地做普通 Java 测试，不涉及 Spring Web 项目，可以省略 @RunWith 注解，这样系统会自动使用默认 Runner 来运行代码。

7.2.3 Spring Boot 单元测试

@SpringBootTest 是 Spring Boot 项目测试的核心注解，标识该测试类以 Spring Boot 方式运行。该注解的源码如下所示。

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@BootstrapWith(SpringBootTestContextBootstrapper.class)
public @interface SpringBootTest {
    @AliasFor("properties")
    String[] value() default {};

    @AliasFor("value")
    String[] properties() default {};

    Class<?>[] classes() default {};

    SpringBootTest.WebEnvironment webEnvironment() default SpringBootTest.
    WebEnvironment.MOCK;
```



```

public static enum WebEnvironment {
    MOCK(false),
    RANDOM_PORT(true),
    DEFINED_PORT(true),
    NONE(false);

    private final boolean embedded;

    private WebEnvironment(boolean embedded) {
        this.embedded = embedded;
    }

    public boolean isEmbedded() {
        return this.embedded;
    }
}

```

可以看到在@SpringBootTest注解源码中最重要的就是@BootstrapWith，该注解才是配置了测试类的启动方式，以及启动时使用实现类的类型。

MockMvc 类是一个被 final 修饰的类型，该类无法被继承使用。这个类是 Spring 为我们提供模拟 SpringMVC 请求的实例类，该类是由 MockMvcBuilders 通过 WebApplicationContext 实例进行创建的，初始化 MockMvc 实例我们可以看一下 before 方法逻辑。编写访问/getUser 请求路径的测试，具体测试代码如下所示。

```

@Test
public void testgetUser() throws Exception {
    MvcResult mvcResult = mockMvc.perform(MockMvcRequestBuilders.get(
        "/getUser").param("id", "372")).andReturn();

    HandlerInterceptor[] interceptors = mvcResult.getInterceptors();
    logger.info(interceptors[0].getClass().getName());

    int status = mvcResult.getResponse().getStatus();
    String responseString = mvcResult.getResponse().getContentAsString();
    logger.info("返回内容: " + responseString);
}

```

```
Assert.assertEquals("return status not equals 200", 200, status);
}
```

(1) perform 方法其实只是为了构建一个请求，并且返回 ResultActions 实例，该实例可以获取到请求的返回内容。

(2) MockMvcRequestBuilders 抽象类可以构建多种请求方式，如 Post、Get、Put、Delete 等常用的请求方式，其中参数是我们需要请求的本项目的相对路径，“/”则是项目请求的根路径。

(3) param 方法用于在发送请求时携带参数，当然除了该方法还有很多其他的方法，可以根据实际请求情况选择调用。

(4) andReturn 方法在发送请求后需要获取返回时调用，该方法返回 MvcResult 对象，该对象可以获取到返回的视图名称、返回的 Response 状态、获取拦截请求的拦截器集合等。

(5) 这里使用了第 4 步中的 MvcResult 对象实例获取的 MockHttpServletResponse 对象，从而才得到的 Status 状态码。

(6) 同样使用 MvcResult 实例获取的 MockHttpServletResponse 对象，从而得到请求返回的字符串内容（可以查看 REST 返回的 JSON 数据）。

(7) 使用 JUnit 内部验证类 Assert 判断返回的状态码是否正常（200）。

(8) 判断返回的字符串是否与我们预计的一样。

```
@Test
public void testInsert()
{
    UserEntity userEntity = new UserEntity();
    userEntity.setNickName("cloudskyme");
    userEntity.setPassWord("cloudskyme");
    userEntity.setUserName("cloud");
    userEntity.setUserSex(UserSexEnum.MAN);
    userMapper.insert(userEntity);
    Assert.assertNotNull(userEntity.getId());
}
```

7.2.4 Mockito

在软件开发的世界之外，“mock”一词是指模仿或者效仿。因此可以将“mock”理解为一

个替身、替代者。在软件开发中提及“mock”，通常理解为模拟对象或者 fake。

Mockito 是一个基于 MIT 协议的开源 Java 测试框架。Mockito 区别于其他模拟框架的地方主要是允许开发者在没有建立“预期”时验证被测系统的行为。对 mock 对象的一个批评是测试代码与被测系统高度耦合，由于 Mockito 试图通过移除“期望规范”来去除 expect-run-verify 模式（期望—运行—验证模式），因此使耦合度降低到最低。这样的突出特性简化了测试代码，使它更容易阅读和修改了。

Spring-boot-starter-test 依赖了 Mockito。

被测类代码如下：

```
public interface Foo {

    boolean checkCodeDuplicate(String code);

}

public interface Bar {

    Set<String> getAllCodes();

}

@Component
public class FooImpl implements Foo {

    private Bar bar;

    @Override
    public boolean checkCodeDuplicate(String code) {
        return bar.getAllCodes().contains(code);
    }

    @Autowired
    public void setBar(Bar bar) {
        this.bar = bar;
    }

}
```

当 Bean 存在这种依赖关系时：LooImpl→FooImpl→Bar，我们应该怎么测试呢？

按照 mock 测试的原则，这时我们应该“mock”一个 Foo 对象，把这个注入 LooImpl 对象里。

不过如果你不想“mock Foo”而是想“mock Bar”的时候，其实做法和前面也差不多，Spring 会自动将 mock Bar 注入 FooImpl 中，然后将 FooImpl 注入 LooImpl 中。

使用 Spring Boot 的测试代码如下：

```
@SpringBootTest(classes = { FooImpl.class })
@TestExecutionListeners(listeners = MockitoTestExecutionListener.class)
public class SpringBootTest extends AbstractTestNGSpringContextTests {

    @MockBean
    private Bar bar;

    @Autowired
    private Foo foo;

    @Test
    public void testCheckCodeDuplicate() throws Exception {

        when(bar.getAllCodes()).thenReturn(Collections.singleton("123"));
        assertEquals(foo.checkCodeDuplicate("123"), true);
    }
}
```

7.3 API 测试

对于 API 的测试，根据不同团队的不同情况，如果测试人员的编码能力强，则建议使用编码的方式进行，方便与持续集成系统进行集成。但是目前能够达到这种级别的测试少之又少，所以测试一般使用工具完成。

Postman 最基础的功能就是发送 HTTP 请求，支持 GET/PUT/POST/DELETE，还有很多其他 HTTP 方法。

通过填写 URL、header、body 等就可以发送一个请求，这对于我们平时做一些简单的测试是够用的。

每次配置完一个请求都可以保存到一个集合中，如此一来，下次测试可以直接从集合中找到你要执行的测试。

集合不单单只有分类和存储功能，Postman 支持一键运行整个集合内的测试。

当然，也可以使用 Hitchhiker。Hitchhiker 是一款开源的 Restful API 测试工具，支持 Schedule、数据对比、压力测试、支持上传脚本定制请求，可以轻松部署到本地，和团队成员一起管理 API。

它与 Postman 的功能对比如表 7-1 所示。

表 7-1 Hitchhiker 和 Postman 对比

功 能	Hitchhiker	Postman
协作性	✓	通过 Share，Pro 收费
脚本	✓ 强，可以上传脚本	✓ 一般，只能用内置的脚本库
Schedule	✓	✓ 需要借助 Newman, Jenkins
数据对比	✓	✗
压力测试	✓	✗
参数化请求	✓	✗
文档	✗ 模板化的文档在计划中	✓，固定格式
API mock	✗	✓
细节，稳定性	一般，待加强	强
安全性	强，本地部署	弱，数据上传

软件界面如图 7-2 所示。

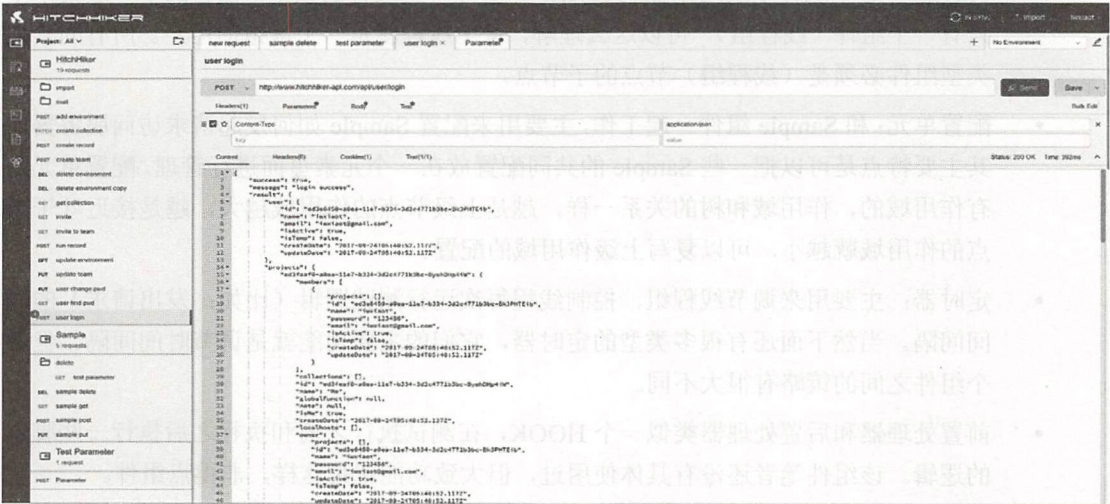


图 7-2 Hitchhiker 软件界面

软件的使用都非常简单，只需要熟悉 HTTP 请求调用，输入对应的请求地址、请求参数，就可以模拟 HTTP 发送请求，完成 API 测试。

7.3.1 Jmeter

Jmeter 是一款专门用于功能测试和压力测试的轻量级测试开发平台。

Jmeter 在概念上的组件分类如图 7-3 所示。

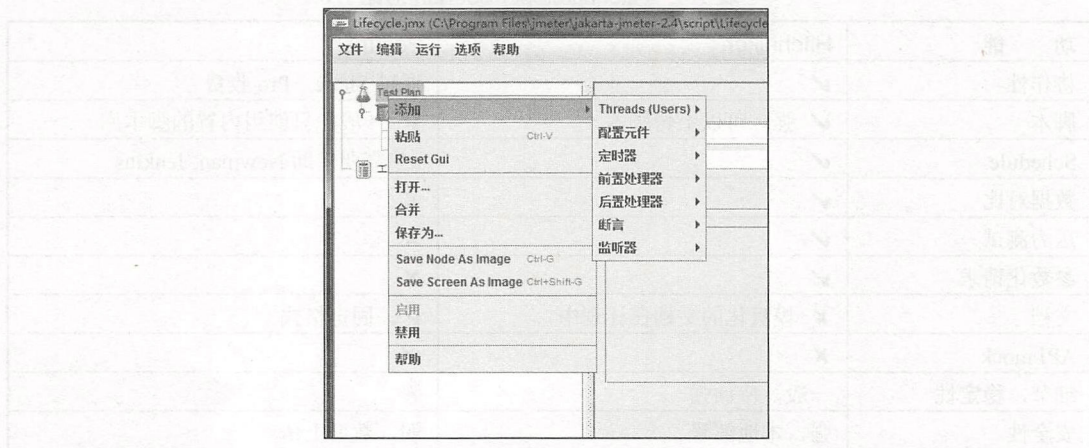


图 7-3 Jmeter 组件分类

- **Threads:** 这个组件主要用来控制 Jmeter 并发时产生线程的数量，在它的下一级菜单下只有一个组件（线程组）。可以这么理解，每个线程就是一个虚拟的用户。所有的其他类型组件必须是（线程组）节点的子节点。
- **配置单元:** 和 Sample 组件一起工作，主要用来配置 Sample 如何发起请求访问服务器，其主要特点是可以把一些 Sample 的共同配置放在一个元素里面进行管理。配置单元是有作用域的，作用域和树的关系一样，越是上级节点的作用域越大，越是接近叶子节点的作用域就越小，可以复写上级作用域的配置。
- **定时器:** 主要用来调节线程组，控制线程每次运行测试逻辑（比如：发出请求）的时间间隔。当然下面还有很多类型的定时器，它们的主要功能就是调节时间间隔，但各个组件之间的策略有很大不同。
- **前置处理器和后置处理器**类似一个 HOOK，在测试执行之前和执行之后执行一些脚本的逻辑。该组件笔者还没有具体使用过，但大致功能就是这样，非重点组件。
- **Sample:** 图 7-3 中没有出现 Sample，需要在（ThreadGroup）上添加才可以，客户端发送某种格式或者规范的请求到服务端，其中有两个 HTTP 相关的。一般用 HttpClient

功能和效率将更强。

- 断言：指 Sample 完成请求发送之后，判断返回的结果是否满足期望。
- 监听器：这个组件不同于平时在 Web 编程的那种监听器，它是伴随 Jmeter 测试的运行而从中抓取运行期间的数据的一个组件，经常使用的是聚合报告组件，从里面可以统计到测试的 TPS、响应时间等关键测试数据。

理解完基本的概念，尝试测试一下 ThreadGroup 组件，设置线程组组件的各项参数，界面如图 7-4 所示。

图 7-4 线程组设置

- 线程数：最大测试时使用的线程数。
- Ramp-Up Period：Jmeter 达到指定最大线程数的时间。
- 循环次数：如果是 Forever，则线程组中的线程将不间断地连续测试系统，当然也可以设置每个线程测试的次数，当完成了规定次数后，该线程将自动退出线程组。
- 调度器：主要用来指定该测试的一些时间信息，比如从几点到几点运行测试，如果到了指定时间测试没有进行完成，则测试也会被停止。

接着在线程组下面添加 Sample 组件，我们添加一个 HTTP Request HTTPClient 组件。最后添加监听器组件：Aggregate Report。

运行后就能够在 report 中间看到一些性能结果的参数。

7.3.2 压力测试

使用 Hitchhiker，进入 Stress 模块，然后单击“create stress test”按钮，在弹出的对话框里填写参数。

- Name: stress test 的名字。
- Collection: 需要做压力测试的 Collection。
- Requests: 选择要执行的请求及排序。
- Repeat: 重复次数。
- Concurrency: 并发数。
- QPS: 每个压力点每秒请求数, 默认为 0, 即没有限制。
- Timeout: 请求的超时时间, 单位为秒, 默认为 0, 即没有限制。
- Keepalive: 请求是否设置 Keepalive。
- Environment: 测试环境。

设置完成后单击“OK”按钮完成创建。

然后使用 Hitchhiker-Node, Hitchhiker-Node 是一个独立的、基于 Golang 的、跨平台的应用程序。

通过下面的链接下载你想要平台的运行文件。

<https://github.com/brookshi/Hitchhiker-Node/releases>

下载完成解压后会有两个文件: Hitchhiker-Node 是一个可执行文件, 也是主体文件, config.json 是一个配置文件, 打开这个配置文件, 把 Address 的值改为部署 Hitchhiker 的 IP, 如 (192.168.0.2:11010)。

现在在计算机上运行这个程序, 它会连接 Hitchhiker Server。当然, 也可以把这个文件放到多台计算机上去执行, 在做压力测试时, 系统会根据这些压力点的 CPU 核数来分配相应的任务, 这样可以支持更大的压力。

压力点准备就绪, 如果已经运行起来, 则可以尝试进行压力测试了。

把鼠标移到 Stress Test item 上, 在弹出的菜单里单击“Run Now”。

Hitchhiker 会显示压力测试实时状况, 包括当前工作的压力点、压力测试的进度、TPS 及请求失败的状态。

效果如图 7-5 所示。

可以看到有 3 个图表 (从上到下):

- 压力测试的当前进度, 包括压力集中在哪些请求上、已经完成的请求数、当前的 TPS。
- 每个请求消耗的时间, 包括 DNS、Connect、Request、Min、Max。
- 请示失败的状态和个数, 失败有 No Response、Server Error(500)、Test Failed 三种。

单击“stop”可以停止压力测试。

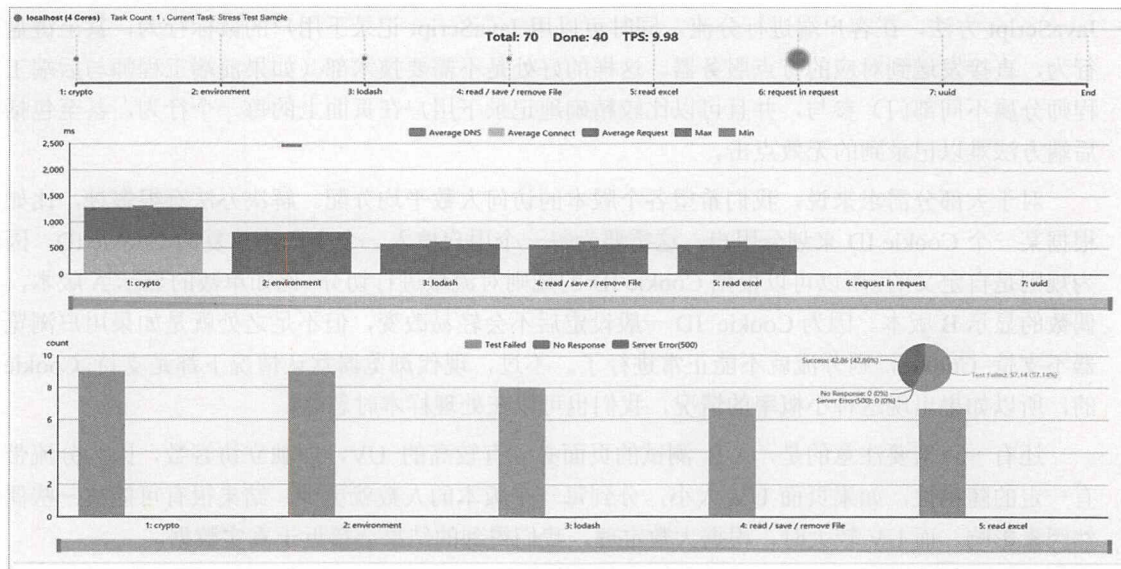


图 7-5 压力测试结果展示

7.4 A/B 测试

A/B 测试又叫分离测试,类似于顾客焦点团体,将一系列内容变化在一定基准内进行比较。A/B 测试来自邮件宣传,发信者将同一目的内容的不同版本邮寄到目标群体中,测量回应率。根据这些数据,商家可以对以后的直邮的内容做相应修改,向更多回应率的版本改进。

A/B 测试最核心的思想,即:

- 多个方案并行测试;
- 每个方案只有一个变量不同;
- 以某种规则优胜劣汰。

事实上,完全可以设计多个方案进行测试,比如 A B C 测试,“A/B 测试”这个名字只是一个习惯的叫法。

A/B 测试需要将多个不同的版本展现给不同的用户,即需要一个“分流”的环节。分流可以在客户端做,也可以在服务器端做。传统的 A/B 测试一般是在服务端分流的,即基于后端的 A/B 测试 (Back-end AB test)。当用户的请求到达服务器时,服务器根据一定的规则,给不同的用户返回不同的版本,同时记录数据的工作也在服务端完成。

基于前端的 A/B 测试的监控的粒度则更细一些,能够定位到具体的用户。利用前端

JavaScript 方法，在客户端进行分流，同时可以用 JavaScript 记录下用户的鼠标行为，甚至键盘行为，直接发送到对应的打点服务器。这样的好处是不需要技术部（如果前端工程师与后端工程师分属不同部门）参与，并且可以比较精确地记录下用户在页面上的每一个行为，甚至包括后端方法难以记录到的无效点击。

对于大部分需求来说，我们希望各个版本的访问人数平均分配。解决办法有很多种，比如根据某一个 Cookie ID 来划分用户，这需要为每一个用户植入一个全局不重复的 Cookie ID，因为规则是自定义的，所以可以根据 Cookie ID 的规则对流量进行切分。比如单数的显示 A 版本，偶数的显示 B 版本。因为 Cookie ID 一般设定后不会轻易改变，但不足之处就是如果用户浏览器不支持 Cookie，则分流就不能正常进行了。不过，现代浏览器默认情况下都是支持 Cookie 的，所以如果出现这种小概率的情况，我们也可以在处理样本时忽略。

还有一点需要注意的是，A/B 测试的页面必须有较高的 UV，即独立访客数，因为分流带有一定的随机性，如果页面 UV 太小，分到每一个版本的人数就更少，结果很有可能被一些偶然因素影响。而 UV 较大时，根据大数定理，我们得到的结果会接近于真实数据。

对于微服务的场景，我们可以使用网关进行分流，分流之后就可以定向采集需要的数据。采集的数据根据业务分析的需要，包括用户的浏览器版本、点击事件的搁置、点击的时间、访问的 URL 信息等。将采集到的数据存储到大数据平台，以供需要分析的部门进行二次加工分析。

7.5 冒烟和回归测试

冒烟测试（Smoke Test）就是在测试中发现问题，找到了一个 Bug，然后开发人员修复这个 Bug。这时想知道这次修复是否真的解决了程序的 Bug，或者是否会对其他模块造成影响，就需要针对此问题进行专门测试，这个过程就被称为 Smoke Test。在很多情况下，做 Smoke Test 是开发人员在试图解决一个问题时，造成了其他功能模块一系列的连锁反应，原因可能是只集中考虑了一开始的那个问题，而忽略其他的问题，这就可能引起了新的 Bug。Smoke Test 的优点是节省测试时间，防止 build 失败，缺点是覆盖率还是比较低。

冒烟测试的名称可以理解为该种测试耗时短，仅用一袋烟功夫足够了。也有人形象地类比为新电路板基本功能检查。任何新电路板焊好后，先通电检查，如果存在设计缺陷，则电路板可能会短路，板子冒烟了。简单地说，就是先保证系统能跑起来，不至于让测试工作做到一半突然出现错误导致业务中断。目的就是先通过最基本的测试，如果最基本的测试都有问题，就直接退回开发部了，减少测试部门时间的浪费。

回归测试是指修改了旧代码后，重新进行测试以确认修改没有引入新的错误或导致其他代码产生错误。自动回归测试将大幅降低系统测试、维护升级等阶段的成本。回归测试作为软件生命周期的一个组成部分，在整个软件测试过程中占有很大的工作量比重，软件开发的各个阶

段都会进行多次回归测试。在渐进和快速迭代开发中，新版本的连续发布使回归测试进行得更加频繁，而在极端编程方法中，更是要求每天都进行若干次回归测试。因此，通过选择正确的回归测试策略来改进回归测试的效率和有效性是非常有意义的。

那么回归测试是否能够自动化呢？答案是肯定的。

所需工具包括：

- Postman——用于编写测试用例，导出.json 格式的测试脚本。
- Newman——命令行方式执行 Postman 导出的测试脚本。
- Jenkins——实现测试的自动化。

测试过程如下：

- 使用 Postman 编写微服务接口的测试用例，并导出 JSON 文件。
- 使用 Jenkins 创建测试项目，使用 newman 来执行测试脚本，生成测试报告。
- 将测试项目与工程构建项目关联，使之在构建发布到测试环境后触发执行。

编写测试用例流程如下：

- 根据 Swagger 服务契约，使用 Postman 工具，传入测试参数并模拟调用。
- 调用后，单击左侧的文件夹图标，新建一个 Collection。
- 单击右侧的 Save 按钮，将本次测试脚本保存到刚才创建的 Collection 中。
- 继续进行其他 API 的调用，并保存至 Collection 中。
- 单击 Collection 上的省略号图标，并选择 Export，将测试用例导出并保存为.json 文件。
- 上传到测试服务器相应目录下。

脚本编写完成后，使用 Jenkins 触发调用 newman 进行测试。相关内容见微服务自动化测试中配置测试一节。

自动化的冒烟和回归测试是非常必要的，能够大幅度地减轻测试团队的工作压力，提高工作效率，但是不可避免的一个问题就是对测试人中的能力要求相应地提高了一些。

7.6 静态代码分析

在软件开发过程中，开发团队往往要花费大量的时间和精力发现并修改代码缺陷。静态代码分析（static code analysis）工具能够在代码构建过程中帮助开发人员快速、有效地定位代码缺陷并及时纠正这些问题，仅通过分析或检查源程序的语法、结构、过程、接口等来检查程序的正确性，找出代码隐藏的 errors 和缺陷，如参数不匹配、有歧义的嵌套语句、错误的递归、非法计算、可能出现的空指针引用等。从而极大地提高软件可靠性并节省软件开发和测试成本。

常用的分析工具包括 Checkstyle、FindBugs、PMD 等，下面我们来一起认识一下这些工具，如表 7-2 所示。

表 7-2 静态代码检查工具

工 具	目 的	检 查 项
FindBugs 检查.class	基于 Bug Patterns 概念，查找 javabytecode（.class 文件）中的潜在 Bug	主要检查 bytecode 中的 bug patterns，如 NullPoint 空指针检查、没有合理关闭资源、字符串相同判断错（==，而不是 equals）等
PMD 检查源文件	检查 Java 源文件中的潜在问题	主要包括： <ul style="list-style-type: none">• 空 try/catch/finally/switch 语句块• 未使用的局部变量、参数和 private 方法• 空 if/while 语句• 过于复杂的表达式，如不必要的 if 语句等• 复杂类
CheckStyle 检查源文件 主要关注格式	检查 Java 源文件是否与代码规范相符	主要包括： <ul style="list-style-type: none">• Javadoc 注释• 命名规范• 多余没用的 Imports• Size 度量，如过长的方法• 缺少必要的空格 Whitespace• 重复代码

7.6.1 Checkstyle

介绍

Checkstyle 是一个开源代码分析工具，能够帮助开发人员保证他们的代码遵循一定的代码规范。Checkstyle 不断地检查代码，一旦发现有违反定义的代码规范的地方就马上提示，以便开发人员能够及时发现和修改不规范代码。

实现

(1) 在需要检查的工程中添加 maven-checkstyle-plugin，代码如下：

```
<reporting>
<plugins>
```



```

<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-checkstyle-plugin</artifactId>
  <version>2.17</version>
</plugin>

<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-jxr-plugin</artifactId>
  <version>2.3</version>
</plugin>
</plugins>
</reporting>

```

maven-checkstyle-plugin 内置了 4 种规范。

- config/sun_checks.xml
- config/maven_checks.xml
- config/turbine_checks.xml
- config/avalon_checks.xml

其中 sun_checks.xml 为默认值。如果想要使用其他三种规范，则只需配置 configuration。

(2) 一般在企业中都会构建自己的质量分析文件，所以我们需要配置自定义的 checkstyle.xml 文件，在资源目录下新建 mycheckstyle.xml，以验证禁止使用 System.out.println 为例。示例代码如下：

```

<!-- 禁止使用 System.out.println -->
<module name="Regexp">
  <property name="format" value="System\.out\.println"/>
  <property name="message" value="不要使用 System.out 与 System.out
进行控制台打印，应该使用日志工具类(如：log4j)进行统一记录或者打印，违反编码规范"/>
  <property name="illegalPattern" value="true"/>
</module>
<!-- 禁止使用 System.err.println -->
<module name="Regexp">
  <property name="format" value="System\.err\.println"/>
  <property name="message" value="不要使用 System.out 与 System.err
进行控制台打印，应该使用日志工具类(如：Log4j)进行统一记录或者打印，违反编码规范"/>

```

```
<property name="illegalPattern" value="true"/>
</module>
```

写一个测试类，在其中加入 `System.out.println`。

测试

执行 `mvn checkstyle:checkstyle` 命令，然后在 `target/site` 目录下生成 `checkstyle.html`，能够查看到生成的报告：

```
Error    regexp    Regexp    Line matches the illegal pattern '不要使用
System.out 与 System.out 进行控制台打印，应该使用日志工具类 (如: log4j) 进行统一记录或者打
印，违反编码规范'.
```

```
Error    regexp    Regexp    Line matches the illegal pattern '不要使用
System.out 与 System.err 进行控制台打印，应该使用日志工具类 (如: log4j) 进行统一记录或者打
印，违反编码规范'.
```

可以根据查看的结果修改相关不符合规则的代码。

Checkstyle 的可执行任务如下：

```
mvn checkstyle:help          查看 checkstyle-plugin 的帮助
mvn checkstyle:check         检查工程是否满足 checkstyle 的检查，如果没有满足，则
检查会失败，可以通过 target/site/checkstyle.html 查看
mvn checkstyle:checkstyle    检查工程是否满足 checkstyle 的检查，如果没有满足，则
检查不会失败，可以通过 target/site/checkstyle.html 查看
mvn checkstyle:checkstyle-aggregate  检查工程是否满足 checkstyle 的检查，如
果没有满足，则检查不会失败，可以通过 target/site/checkstyle.html 查看
```

总结

CheckStyle 能够帮助程序员检查代码是否符合制定的规范。通过将 CheckStyle 的检查引入到项目构建中，可以强制让项目中的所有开发者遵循制定规范，而不是仅仅停留在纸面上。如果发现代码违反了标准，比如类名未以大写开头、单个方法超过了指定行数、甚至单个方法抛出了 3 个以上的异常等 CheckStyle 能够给出相应的提示，提醒开发人员处理。这些检查由于是基于源码的，所以不需要编译，执行速度非常快。

7.6.2 FindBugs

介绍

FindBugs 是由马里兰大学提供的一款开源 Java 静态代码分析工具。FindBugs 通过检查类文件或 JAR 文件，将字节码与一组缺陷模式进行对比从而发现代码缺陷，完成静态代码分析。FindBugs 既提供可视化 UI 界面，也可以作为 Eclipse 插件使用。本节主要使用 FindBugs 作为 Eclipse 插件。在安装成功后会在 Eclipse 中增加 FindBugs perspective，用户可以对指定 Java 类或 JAR 文件运行 FindBugs，此时 FindBugs 会遍历指定文件，进行静态代码分析。

实现

(1) 在 pom.xml 中添加 Maven 插件，使用最新的 FindBugs。如果想要生成 HTML 报告，则需要将插件放在 reporting 标签中。

```
<build>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>findbugs-maven-plugin</artifactId>
      <version>2.5.1</version>
      <configuration>
        <threshold>High</threshold>
        <effort>Default</effort>
        <findbugsXmlOutput>true</findbugsXmlOutput>
        <!-- findbugs xml 输出路径-->
        <findbugsXmlOutputDirectory>target/site</findbugsXmlOutputDirectory>
      </configuration>
    </plugin>
  </plugins>
</build>
```

FindBugs 基于 Bug Patterns 概念，查找 javabytecode (.class 文件) 中的潜在 Bug，主要检查 bytecode 中的 bug patterns，如 NullPoint 空指针检查、没有合理关闭资源、字符串相同判断错 (==，而不是 equals) 等。

FindBugs 的相关命令如下：

```
mvn findbugs:help
```

查看 FindBugs 插件的帮助

```
mvn findbugs:check      检查代码是否通过 FindBugs 检查，如果没有通过检查，则检查会失败，但检查不会生成结果报表
mvn findbugs:findbugs   检查代码是否通过 FindBugs 检查，如果没有通过检查，则检查不会失败，会生成结果报表保存在 target/findbugsXml.xml 文件中
mvn findbugs:gui        检查代码并启动 GUI 界面来查看结果
```

测试

- (1) 执行 `mvn compile findbugs:findbugs`，这里一定要先编译，因为 FindBugs 是检查 class 文件。
- (2) 执行 `mvn compile site` 生成报告。
- (3) 在 `target/site/findbugs.html` 中查看生成的结果。

总结

使用 Maven 的方式集成 FindBugs 非常方便，这也为我们做自动化的质量管理集成带来了便利。也就是说，当我们每次提交代码时都能够重新构建工程，然后有针对性地生成质量检查报告。

7.6.3 PMD

介绍

PMD 是一种开源分析 Java 代码错误的工具。与其他分析工具不同的是，PMD 通过静态分析获知代码错误。也就是说，在不运行 Java 程序的情况下报告错误。PMD 附带了许多可以直接使用的规则，利用这些规则可以找出 Java 源程序的许多问题。

- 潜在的 Bug：空的 `try/catch/finally/switch` 语句。
- 未使用的代码：未使用的局部变量、参数、私有方法等。
- 可选的代码：String/StringBuffer 的滥用。
- 复杂的表达式：不必需的 `if` 语句、可以使用 `while` 循环完成的 `for` 循环。
- 重复的代码：复制/粘贴代码意味着复制/粘贴 bugs。
- 循环体创建新对象：尽量不要在 `for` 或 `while` 循环体内实例化一个新对象。
- 资源关闭：Connect、Result、Statement 等使用之后确保关闭。

此外，用户还可以自己定义规则，检查 Java 代码是否符合某些特定的编码规范。例如，可以编写一个规则，要求 PMD 找出所有创建 Thread 和 Socket 对象的操作。

原理

PMD 是一种代码静态分析工具,当使用 PMD 规则分析 Java 源码时,PMD 首先利用 JavaCC 和 EBNF 文法产生了一个语法分析器,用来分析普通文本形式的 Java 代码,产生符合特定语法结构的语法。同时在 JavaCC 的基础上添加了语义的概念即 JJTree,通过 JJTree 的一次转换,这样就将 Java 代码转换成了一个 AST,AST 是 Java 符号流之上的语义层,PMD 把 AST 处理成一个符号表。然后编写 PMD 规则,一个 PMD 规则可以看作一个 Visitor,通过遍历 AST 找出多个对象之间的一种特定模式,即代码所存在的问题。

自定义 PMD 实现规则有如下 2 种方式:

(1) 自定义 Java 类并继承 `AbstractJavaRule` 抽象类,重写 `visit()` 方法,并在 XML 规则文件中引用该类。

- `name`: 自定义规则的名字;
- `language`: 要检查的语言;
- `message`: 该规则被触发时给出的消息提示;
- `class`: 规则使用类的全类名;
- `description`: 规则的描述信息;
- `priority`: 优先级别,从高到低依次是 1-Blocker、2-Critical、3-Urgent、4-important、5-Warning;
- `example`: 在 CDATA 标签中书写一个该规则对应的实例。

(2) 自定义 XPath 表达式,编写 XML 规则,在规则的 `properties-property` 节点中定义 XPath 表达式,该表达式是依赖于抽象语法树 AST 的。

- `name`: 自定义规则的名字;
- `language`: 要检查的语言;
- `message`: 该规则被触发时给出的消息提示;
- `class`: XPath 规则统一配置为 `net.sourceforge.pmd.lang.rule.XPathRule`;
- `description`: 规则的描述信息;
- `priority`: 优先级别,从高到低依次是 1-Blocker、2-Critical、3-Urgent、4-important、5-Warning;
- `example`: 在 CDATA 标签中书写一个该规则对应的实例;
- `properties`: 这个是 XPath 必须配置的,其子节点为 `property`,`value` 值使用 CDATA 标签配置对应的 XPath 表达式的形式,可以有多个 `property`。

实现

在 pom.xml 中添加相关依赖，代码如下：

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-pmd-plugin</artifactId>
      <version>3.8</version>
      <configuration>
        <rulesets>
          <ruleset>rulesets/java/ali-comment.xml</ruleset>
          <ruleset>rulesets/java/ali-concurrent.xml</ruleset>
          <ruleset>rulesets/java/ali-constant.xml</ruleset>
          <ruleset>rulesets/java/ali-exception.xml</ruleset>
          <ruleset>rulesets/java/ali-flowcontrol.xml</ruleset>
          <ruleset>rulesets/java/ali-naming.xml</ruleset>
          <ruleset>rulesets/java/ali-oop.xml</ruleset>
          <ruleset>rulesets/java/ali-orm.xml</ruleset>
          <ruleset>rulesets/java/ali-other.xml</ruleset>
          <ruleset>rulesets/java/ali-set.xml</ruleset>
        </rulesets>
        <printFailingErrors>true</printFailingErrors>
      </configuration>
    <executions>
      <execution>
        <goals>
          <goal>check</goal>
        </goals>
      </execution>
    </executions>
  <dependencies>
    <dependency>
      <groupId>com.alibaba.p3c</groupId>
      <artifactId>p3c-pmd</artifactId>
      <version>1.3.0</version>
    </dependency>
  </dependencies>
</build>
```



```
</plugin>
</plugins>
</build>
```

在代码中看到，我们集成的是 p3c-pmd，p3c-pmd 插件的实现是基于 PMD 的，具体来说是基于 pmd-java 的，因为 PMD 不仅支持 Java 代码分析，还支持其他多种语言。

其中，阿里实现的 PMD 的开源地址为：<https://github.com/alibaba/p3c>。

可见顶级公司对于代码质量的管理非常重视。

总结

在代码开发完成之后，就需要对其质量进行监控，防止在线上环境出现严重的后果。

7.7 SonarQube 质量监控

Sonar (SonarQube) 是一个开源平台，用于管理源代码的质量。Sonar 不只是一个质量数据报告工具，更是代码质量管理平台。通过插件机制，Sonar 可以集成不同的测试工具，代码分析工具，以及持续集成工具。

与持续集成工具（例如，Hudson/Jenkins 等）不同，Sonar 并不是简单地把不同的代码检查工具（例如，FindBugs、PMD 等）结果直接显示在 Web 页面上，而是通过不同的插件对这些结果进行再加工处理，通过量化的方式度量代码质量的变化，从而可以方便地对不同规模和种类的工程进行代码质量管理。

在对其他工具的支持方面，Sonar 不仅提供了对 IDE 的支持，可以在 Eclipse 和 IntelliJ IDEA 这些工具里联机查看结果；同时 Sonar 对大量的持续集成工具提供了接口支持，可以很方便地在持续集成中使用 Sonar。

此外，Sonar (SonarQube) 的插件还可以对 Java 以外的其他编程语言提供支持，对国际化及报告文档化也有良好的支持，支持的语言包括 Java、PHP、C#、C、Cobol、PL/SQL、Flex 等。

7.7.1 为什么使用

SonarQube 可以从多个维度检测代码质量，包括以下维度。

- (1) 复杂度分布 (complexity)：代码复杂度过高将难以理解、难以维护。
- (2) 重复代码 (duplications)：程序中包含大量复制粘贴的代码是质量低下的表现。
- (3) 单元测试 (unit tests)：统计并展示单元测试覆盖率。

- (4) 编码规范 (coding rules): 通过 FindBugs、PMD、CheckStyle 等规范代码编写。
- (5) 注释 (comments): 少了可读性差, 多了看起来费劲。
- (6) 潜在的 Bug (potential bugs): 通过 FindBugs、PMD、CheckStyle 等检测潜在的 Bug。
- (7) 结构与设计 (architecture & design): 依赖、耦合等。

7.7.2 安装和使用

为了保证对系统的无侵入性, 我们使用 Docker 的方式进行安装。

使用 docker-compose.yml 文件:

```
version: '2'

services:
  sonarqube:
    image: sonarqube
    container_name: sonarqube-server
    ports:
      - "9000:9000"
      - "5432:5432"
    links:
      - db:db
    environment:
      - SONARQUBE_JDBC_URL=jdbc:postgresql://db:5432/sonar
    volumes:
      - /tmp:/opt/sonarqube/extensions

  db:
    image: postgres
    container_name: postgres
    environment:
      - POSTGRES_USER=sonar
      - POSTGRES_PASSWORD=sonar
```

然后执行构建:

```
docker-compose up -d
```


构建并启动完成后, 打开浏览器输入网址 `http://IP:9000`, 默认的用户名是 `admin`, 密码是 `admin`。

分析项目使用的是 `sonarqube scanner`, 可以使用它的客户端版本, 也可以使用 Maven 插件的方式。

使用 Maven 插件的方式是在配置中加入如下代码:

```
<profile>
  <id>sonar</id>
  <activation>
    <activeByDefault>true</activeByDefault>
  </activation>
  <properties>
    <!-- Optional URL to server. Default value is http://localhost: 9000 -->
    <sonar.host.url>
      http://IP:9000
    </sonar.host.url>
  </properties>
</profile>
```

然后在控制台检查 Maven 项目, 在项目目录下运行:

```
mvn sonar:sonar
```

运行完成后, 我们能够在构建的工程工作台上看到质量检查的结果, 界面如图 7-6 所示。

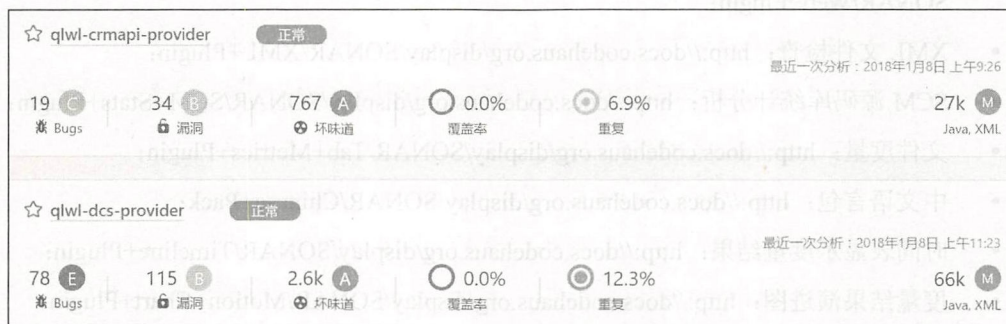


图 7-6 sonar 质量检查结果

图 7-6 的运行结果是根据 Sonar 自身带的校验规则完成的检查, 每个公司的情况都不一样, 所以建议使用它的定制功能, 缩写适合自己公司的代码质量检查规则。

7.7.3 安装插件

Sonar 支持多种插件，插件的下载地址为 <http://docs.codehaus.org/display/SONAR/Plugin+Library>。

将下载后的插件上传到 `${SONAR_HOME}\extensions\plugins` 目录下，重新启动 Sonar。

Sonar 默认集成了 Java Ecosystem 插件，该插件是一组插件的合集。

- Java [sonar-java-plugin]: Java 源代码解析，计算指标等；
- Squid [sonar-squid-java-plugin]: 检查违反 Sonar 定义规则的代码；
- Checkstyle [sonar-checkstyle-plugin]: 使用 CheckStyle 检查违反统一代码编写风格的代码；
- FindBugs [sonar-findbugs-plugin]: 使用 FindBugs 检查违反规则的缺陷代码；
- PMD [sonar-pmd-plugin]: 使用 PMD 检查违反规则的代码；
- Surefire [sonar-surefire-plugin]: 使用 Surefire 执行单元测试；
- Cobertura [sonar-cobertura-plugin]: 使用 Cobertura 获取代码覆盖率；
- JaCoCo [sonar-jacoco-plugin]: 使用 JaCoCo 获取代码覆盖率。

下面列出了一些常用的插件地址。

- JavaScript 代码检查: <http://docs.codehaus.org/display/SONAR/JavaScript+Plugin>;
- Python 代码检查: <http://docs.codehaus.org/display/SONAR/Python+Plugin>;
- Web 页面检查 (HTML、JSP、JSF、Ruby、PHP 等): <http://docs.codehaus.org/display/SONAR/Web+Plugin>;
- XML 文件检查: <http://docs.codehaus.org/display/SONAR/XML+Plugin>;
- SCM 源码库统计分析: <http://docs.codehaus.org/display/SONAR/SCM+Stats+Plugin>;
- 文件度量: <http://docs.codehaus.org/display/SONAR/Tab+Metrics+Plugin>;
- 中文语言包: <http://docs.codehaus.org/display/SONAR/Chinese+Pack>;
- 时间表显示度量结果: <http://docs.codehaus.org/display/SONAR/Timeline+Plugin>;
- 度量结果演进图: <http://docs.codehaus.org/display/SONAR/Motion+Chart+Plugin>。

7.7.4 运行流程

整个源代码的质量检查流程如图 7-7 所示。

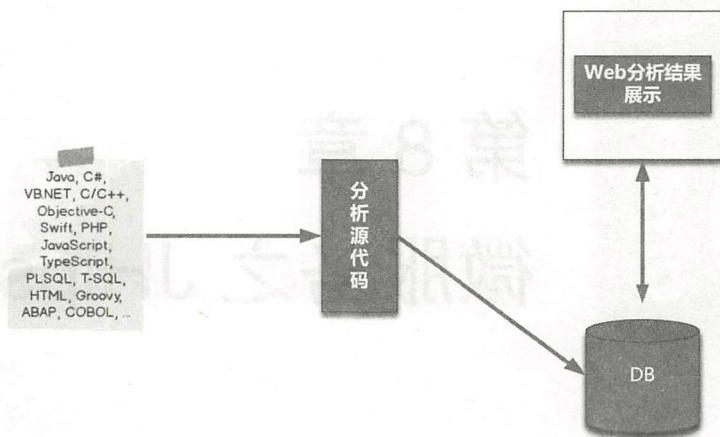


图 7-7 质量检查运行流程

首先从 SVN 或者 Git 代码仓库中获取静态代码。

对静态代码进行扫描分析，根据事先定义好的规则扫描出代码中可能存在的漏洞。

把扫描的结果数据存储到数据库中。使用可视化的界面进行分析和展示。

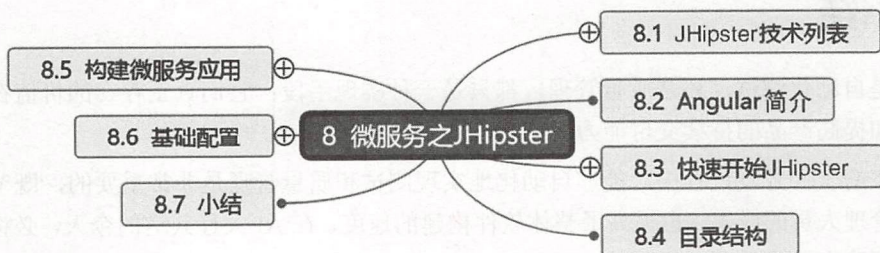
7.8 小结

不论是自动化测试，还是质量管理，都只是一种实现手段；它们真正存在的价值在于提高代码质量和提高产品的持续交付能力。

在现在的软件开发过程中，能够自动化地实现测试和质量管理的非常重要的，既节省了测试和质量管理人员的成本，也提高了整体软件构建的速度，在 AI 大行其道的今天，必将成为微服务生态闭环中非常重要一个环节。

8 chapter

第 8 章 微服务之 JHipster



代码生成器是按照特定编码规范输出代码的软件，可以直接生成项目，也可以单页生成，自从编程开始就有很多人进行过各种尝试。国内也有很多这种快速开发的框架，但是这些框架除了个别的开源项目，往往持续性不够，不能够及时跟上技术发展的步伐。而且，对于代码的质量也没有进行很好的监控，没有一套完整的方案。那么，在微服务大行其道的当下，是不是有一款适合构建微服务的代码生成框架呢？答案就是 JHipster。

JHipster 官网的原文介绍：

JHipster is a development platform to generate, develop and deploy Spring Boot + Angular Web applications and Spring microservices.

JHipster 是一个开发框架，用于生成、开发和部署基于 Spring Boot 和 AngularJS 的 Web 工程和基于 Spring 的微服务工程。

Spring Boot 在前面已经介绍过，相信通过源码加注解，我们已经能够体会到它的强大了，那么 AngularJS 是什么呢？简单地说就是一套 JavaScript 的工具包，或者说是一个 JavaScript 框架。它通过为开发者呈现一个更高层次的抽象来简化应用的开发。同其他的抽象技术一样，这会损失一部分灵活性。换句话说，并不是所有的应用都适合用它来开发。AngularJS 主要考虑的是构建 CRUD 应用。幸运的是，至少 90% 的 Web 应用都是 CRUD 应用。那么哪些应用不适合使用它呢？比如游戏、图形界面编辑器等这种 DOM 操作很频繁也很复杂的应用，和 CRUD 应用就有很大的不同，用一些更轻量、更简单的技术如 jQuery 可能会更好。

8.1 JHipster 技术列表

在使用一项技术前，就需要了解这项技术应用的一些特性，以及它所包含的技术列表。JHipster 能够将多项技术无缝地整合在一起，构建出强大的应用。

8.1.1 客户端选项

客户端集成的技术如图 8-1 所示。



图 8-1 客户端技术列表

- **HTML 5:** HTML 5 是最新一代的 HTML 标准，它不仅拥有 HTML 中所有的特性，而且增加了许多实用的特性，如视频、音频、画布（Canvas）等。
- **CSS 3:** CSS 3 是 CSS 技术的升级版本，CSS 3 语言开发是朝着模块化发展的。以前的规范作为一个模块实在太庞大，而且比较复杂，所以把它分解为一些小的模块，更多新的模块也被加入进来。这些模块包括盒子模型、列表模块、超链接方式、语言模块、背景和边框、文字特效、多栏布局等。
- **Bootstrap:** Bootstrap 来自 Twitter，是目前最受欢迎的前端框架。Bootstrap 是基于 HTML、CSS、JavaScript 的，它简洁灵活，使得 Web 开发更加快捷。
- **AngularJS:** AngularJS 诞生于 2009 年，由 Misko Hevery 等人创建，后为 Google 所收购，是一款优秀的前端 JS 框架，已经被用于 Google 的多款产品当中。AngularJS 有着诸多优秀的特性，最为核心的是：MVC、模块化、自动化双向数据绑定、语义化标签、依赖注入等。
- **Angular:** 针对 AngularJS 重新设计的前端框架。
- **JQuery:** 高度封装的 JavaScript 库，简化前端开发。
- **Websockets:** 能够让用户在浏览器上实现 socket 功能，用于实时推送等场景。
- **Yarn:** 新一代前端框架。
- **Webpack:** Webpack 可以看作模块打包机：它做的事情是分析你的项目结构，找到 JavaScript 模块及其他的一些浏览器不能直接运行的拓展语言（SCSS、TypeScript 等），

并将其转换和打包为合适的格式供浏览器使用。

- **Bower:** Bower 是一个客户端技术的软件包管理器，它可用于搜索、安装和卸载如 JavaScript、HTML、CSS 之类的网络资源。其他一些建立在 Bower 基础之上的开发工具有 Yeoman 和 Grunt。
- **Gulp:** Gulp 是前端开发过程中对代码进行构建的工具，是自动化项目的构建利器；它不仅能对网站资源进行优化，而且在开发过程中很多重复的任务能够使用正确的工具自动完成；使用它，我们不仅可以很愉快地编写代码，而且大大提高工作效率。
- **Sass:** Sass 是对 CSS 的扩展，让 CSS 语言更强大、优雅。它允许你使用变量、嵌套规则、mixins、导入等众多功能，并且完全兼容 CSS 语法。
- **Browsersync:** Browsersync 能让浏览器实时、快速响应文件更改（HTML、JS、CSS、Sass、Less 等）并自动刷新页面。更重要的是 Browsersync 可以同时 PC、平板电脑、手机等设备下进项调试。可以想象一下：“假设桌子上有 PC、iPad、iPhone、Android 等设备，同时打开了需要调试的页面，当使用 Browsersync 后，任何一次代码保存，以上的设备都会同时显示改动”。无论是前端还是后端工程师，使用它将提高 30% 的工作效率。
- **Karma:** Karma 是由 Google 团队开发的一套前端测试运行框架。它不同于测试框架（例如，Jasmine、Mocha 等），运行在这些测试框架之上，主要完成以下工作：
 - Karma 启动一个 Web 服务器，生成包含 JS 源代码和 JS 测试脚本的页面；
 - 运行浏览器加载页面，并显示测试的结果；
 - 如果开启检测，则当文件有修改时，继续执行以上过程。
- **Protractor:** Protractor 是 AngularJS 团队发布的一款开源的端到端 Web 测试运行工具。它可以模拟用户的实际交互，帮助验证 Angular 应用的实际运行状况。Protractor 使用 Jasmine 测试框架来定义测试用例。Protractor 为不同的页面交互提供一套健壮的 API。相对于其他的端到端的工具，Protractor 有着自己的优势，它知道怎么和 AngularJS 的代码一起运行，特别是应对 \$digest 循环。

8.1.2 服务端选项

服务端集成的技术如图 8-2 所示。

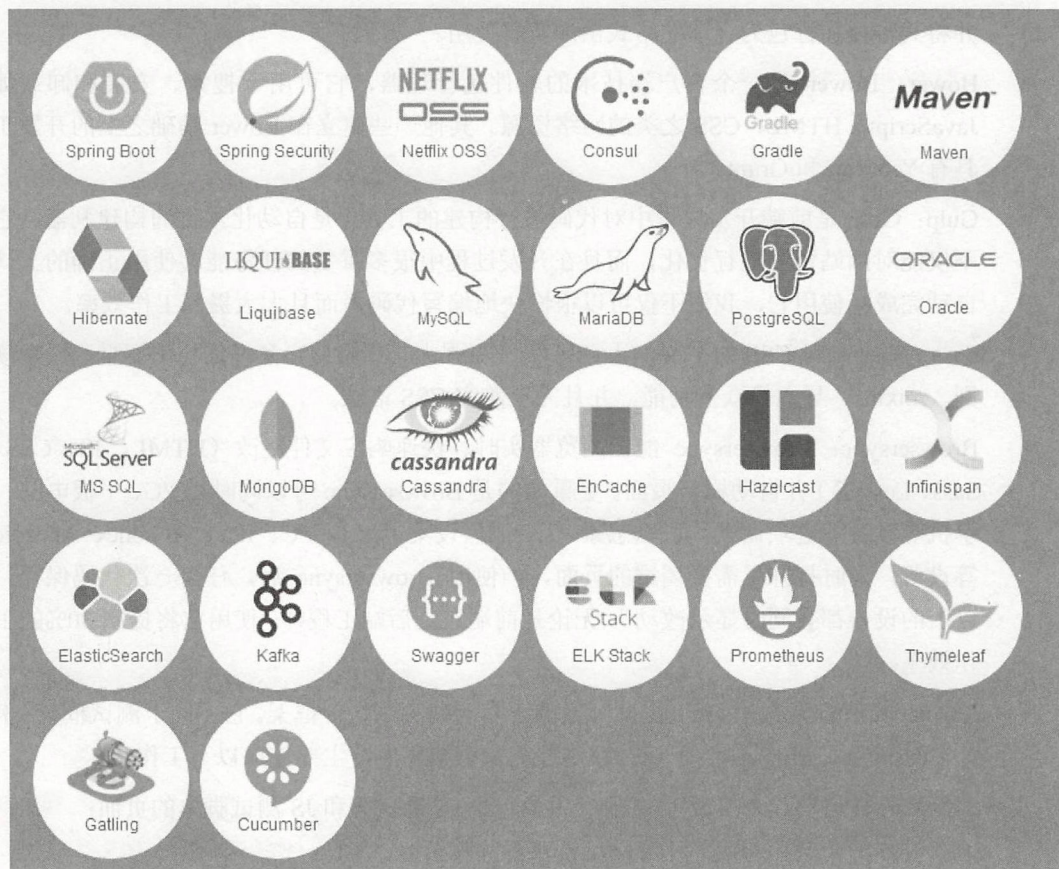


图 8-2 服务端技术列表

- **Spring Boot:** 内容见专门讲解 Spring Boot 的章节。
- **Spring Security:** Spring Security 是一个能够为基于 Spring 的企业应用系统提供声明式的安全访问控制解决方案的安全框架。它提供了一组可以在 Spring 应用上下文中配置的 Bean, 充分利用了 Spring IoC、DI (控制反转, Inversion of Control; DI: Dependency Injection, 依赖注入) 和 AOP (面向切面编程) 功能, 为应用系统提供声明式的安全访问控制功能, 减少了为企业系统安全控制而编写大量重复代码的工作。
- **Netflix OSS:** Netflix OSS 是由 Netflix 公司主持开发的一套代码框架和库, 目的是解决规模大了之后的分布式系统可能出现的一些问题。
- **Consul:** Consul 是一个支持多数据中心分布式高可用的服务发现和配置共享的服务软件, 由 HashiCorp 公司用 Go 语言开发, 基于 Mozilla Public License 2.0 的协议进行开源。Consul 支持健康检查, 并允许 HTTP 和 DNS 协议调用 API 存储键值对。

- **Gradle:** Gradle 是一个基于 Apache Ant 和 Apache Maven 概念的项目自动化构建工具。它使用一种基于 Groovy 的特定领域语言 (DSL) 来声明项目设置, 抛弃了基于 XML 的各种烦琐配置。面向 Java 应用为主。当前其支持的语言限于 Java、Groovy、Kotlin 和 Scala, 计划未来将支持更多的语言。
- **Maven:** Maven 是一个项目管理工具, 它包含了一个项目对象模型 (Project Object Model)、一组标准集合、一个项目生命周期 (Project Lifecycle)、一个依赖管理系统 (Dependency Management System) 和用来运行定义在生命周期阶段 (phase) 中插件 (plugin) 目标 (goal) 的逻辑。当使用 Maven 时, 用一个明确定义的项目对象模型来描述项目, 然后 Maven 可以应用横切的逻辑, 这些逻辑来自一组共享的 (或者自定义的) 插件。
- **Hibernate:** Hibernate 是一个开放源代码的对象关系映射框架, 它对 JDBC 进行了非常轻量级的对象封装, 它将 POJO 与数据库表建立映射关系, 是一个全自动的 ORM 框架, Hibernate 可以自动生成 SQL 语句, 自动执行, 使得 Java 程序员可以随心所欲地使用对象编程思维来操纵数据库。
- **Liquibase:** Liquibase 是一个用于跟踪、管理和应用数据库变化的开源的数据库重构工具。它将所有数据库的变化 (包括结构和数据) 都保存在 XML 文件中, 便于版本控制。
- **MySQL:** MySQL 是一个关系型数据库管理系统, 由瑞典 MySQL AB 公司开发, 目前属于 Oracle 旗下产品。MySQL 是最流行的关系型数据库管理系统之一, 在 Web 应用方面, MySQL 是最好的 RDBMS (Relational Database Management System, 关系数据库管理系统) 应用软件之一。
- **MariaDB:** MariaDB 数据库管理系统是 MySQL 的一个分支, 主要由开源社区在维护, 采用 GPL 授权许可 MariaDB 的目的是完全兼容 MySQL, 包括 API 和命令行, 使之能轻松成为 MySQL 的替代品。
- **PostgreSQL:** PostgreSQL 是一个功能强大的开源对象关系数据库管理系统 (ORDBMS)。用于安全地存储数据; 支持最佳做法, 并允许在处理请求时检索它们。
- **Oracle:** Oracle Database, 又名 Oracle RDBMS, 或简称 Oracle, 是甲骨文公司的一款关系数据库管理系统。它是在数据库领域一直处于领先地位的产品。可以说 Oracle 数据库系统是目前世界上流行的关系数据库管理系统, 系统可移植性好、使用方便、功能强, 适用于各类大、中、小、微机环境。它是一种高效率、可靠性好的适应高吞吐量的数据库解决方案。
- **MSSQL:** Microsoft SQL Server 是一个数据库平台, 其数据库引擎为关系型数据和结构化数据提供了更安全可靠存储功能, 可以构建和管理用于业务的高可用和高性能的数据应用程序。

- **MongoDB:** MongoDB 是一个基于分布式文件存储的数据库，由 C++ 语言编写。旨在为 Web 应用提供可扩展的高性能数据存储解决方案。MongoDB 是一个介于关系数据库和非关系数据库之间的产品，是非关系数据库当中功能最丰富、最像关系数据库的。
- **Cassandra:** Cassandra 是一个来自 Apache 的分布式数据库，具有高度可扩展性，可用于管理大量的结构化数据。它提供了高可用性，没有单点故障。
- **EhCache:** EhCache 是一个纯 Java 的进程内缓存框架，具有快速、精干等特点，是 Hibernate 中默认的 CacheProvider。
- **Hazelcast:** Hazelcast 是由 Hazelcast 公司（没错，这家公司也叫 Hazelcast）开发和维护的开源产品，可以为基于 JVM 环境运行的各种应用提供分布式集群和分布式缓存服务。
- **Infinispan:** Infinispan 是一个在 Apache 2.0 开源协议下所开发的，基于内存来进行键值对存储的分布式存储工具，并且其数据格式可以完全自定义。它既可以作为一个 Java 库进行使用，也可以通过一系列主流的远程协议方式（HotRod、REST、Memcached 和 WebSockets）来提供独立的服务。与此同时，它还支持很多高级功能，诸如事务机制、事件机制、查询及分布式处理等。
- **Elasticsearch:** Elasticsearch 是一个基于 Lucene 的搜索服务器。它提供了一个分布式多用户能力的全文搜索引擎，基于 RESTful Web 接口。Elasticsearch 是用 Java 开发的，并作为 Apache 许可条款下的开放源码发布，是当前流行的企业级搜索引擎，设计用于云计算中，能够达到实时搜索，稳定、可靠、快速，安装使用方便。
- **Kafka:** Kafka 是一种高吞吐量的分布式发布订阅消息系统，它可以处理消费者规模的网站中的所有动作流数据。
- **Swagger:** Swagger 是一款 RESTFUL 接口的文档在线自动生成+功能测试功能软件，是一个规范和完整的框架，用于生成、描述、调用和可视化 RESTful 风格的 Web 服务。
- **ELK Stack:** ELK 是 Elasticsearch、Logstash 和 Kibana 的简称。关于 ELK 本书有章节专门介绍。
- **Prometheus:** Prometheus 是一个开源的系统监控和报警的工具包，最初由 SoundCloud 发布。
- **Thymeleaf:** Thymeleaf 是一个 XML/XHTML/HTML5 模板引擎，可以用于 Web 与非 Web 应用。本书关于 Spring Boot 的 Web 开发就是与 Thymeleaf 整合的。
- **Gatling:** Gatling 是一个使用 Scala 编写的开源的负载测试框架，基于 Akka 和 Netty。
- **Cucumber:** Cucumber 是一个能够理解用普通语言描述的测试用例的行为驱动开发（BDD）的自动化测试工具，用 Ruby 编写，支持 Java 和 .NET 等多种开发语言。

8.1.3 部署选项

环境部署集成的技术如图 8-3 所示。

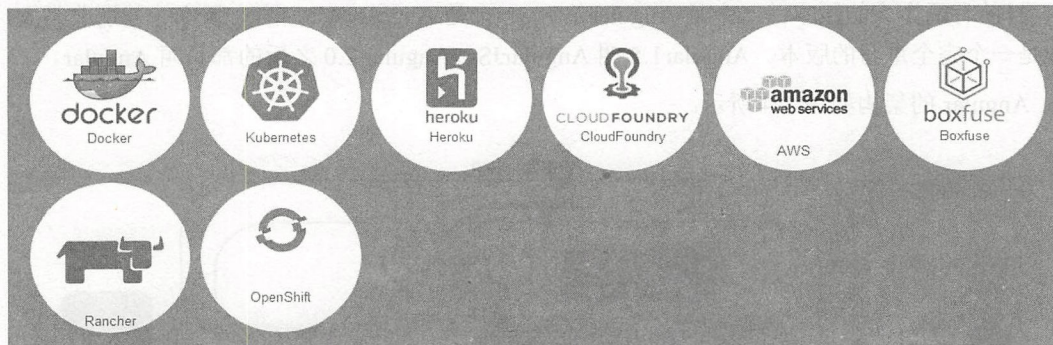


图 8-3 部署环境技术列表

- **Docker:** Docker 是一个开源的应用容器引擎，让开发者可以打包他们的应用和依赖包到一个可移植的容器中，然后发布到任何流行的 Linux 机器上，也可以实现虚拟化，容器是完全使用沙箱机制的，相互之间不会有任何接口。
- **Kubernetes:** Kubernetes (k8s) 是自动化容器操作的开源平台，这些操作包括部署、调度和节点集群间扩展。
- **Heroku:** 开源 PaaS 平台，支持多语言。
- **Cloud Foundry:** Cloud Foundry 是 VMware 推出的业界第一个开源 PaaS 云平台，它支持多种框架、语言、运行时环境、云平台及应用服务，使开发人员能够在几秒钟内进行应用程序的部署和扩展，无须担心任何基础架构的问题。
- **AWS:** 亚马逊提供的云平台服务，国内也有阿里云等云平台厂商。
- **Boxfuse:** Boxfuse 公司提供的一种打包工具，可以快速打包应用到 AWS 上。
- **Rancher:** Rancher 是一个用于部署和管理生产环境的容器的开源平台，它与 Kubernetes/Mesos/Docker Swarm 进行集成，使得在任何硬件环境上容器化应用变得触手可及。
- **OpenShift:** OpenShift 是红帽公司推出的一个云计算服务平台，配置自己的 OpenShift 可以通过几种方式，比如 Web 端、命令行。

看到这些新技术，你的第一反应是什么呢？是不是很兴奋！是不是摩拳擦掌，准备进入学习的状态了呢？

8.2 Angular 简介

Angular 是由谷歌公司维护的一个开源 JavaScript 框架。

目前有两个大的版本：一个是 Angular 1.5，一个是 Angular 4.0。两个版本的差别非常大，4.0 是一个完全重写的版本。Angular 1.5 叫 AngularJS，Angular 2.0 之后的都只叫 Angular。

Angular 的架构如图 8-4 所示。

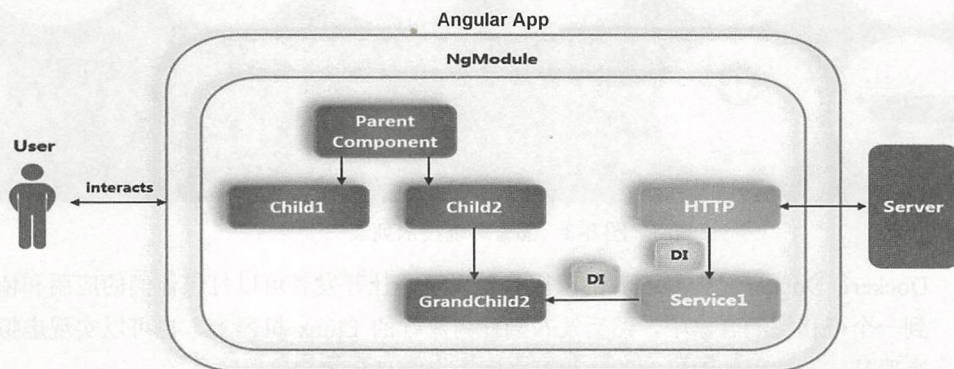


图 8-4 Angular 架构

这个架构图展现了 Angular 应用中的主要构造块。

➤ 模块（module）

模块用来将应用中不同的部分组织成一个 Angular 框架可以理解的单元。

➤ 组件（component）

组件是 Angular 应用的基本构建块，可以把一个组件理解为一块带有业务逻辑和数据的 HTML。

组件的必备要素包括装饰器、模板和控制器。模板显示的内容通过数据绑定进行数据的交互。

➤ 指令（directive）

允许你向 HTML 元素添加自定义行为。

➤ 服务（service）

用来封装可重用的业务逻辑。

➤ 依赖注入（dependency injection）

依赖注入是提供类的新实例的一种方式，还负责处理类所需的全部依赖。大多数依赖都是

服务。Angular 使用依赖注入来提供新组件及组件所需的服务。

通过以上的架构就可以简单地了解 Angular，其实如果是想从架构上面了解，则 Angular 可以理解成一个前端框架，通过 Node.js 运行，完成与后台的交互，它本身基于 MVC 架构，能够高效地实现前端的复杂交互。

8.3 快速开始 JHipster

学习各种软件都是从安装开始的。JHipster 的开源地址为 <https://github.com/jhipster>。

8.3.1 安装

JHipster 的安装和使用方式有以下几种。

➤ 在线使用

地址为 <https://start.jhipster.tech/#/>。

在右上角注册并且绑定 GitHub，就可以直接用在线的方式生成相应的程序代码，然后从 GitHub 上获取生成后的代码信息。这种方式可以快速开始熟悉和使用 JHipster，不需要在本地安装任何东西。

本地安装使用 Yarn:

- (1) 安装 JDK8。
- (2) 安装 node.js，网址 <https://nodejs.org/en/>，安装 LTS 版本。
- (3) 安装 Yarn，网址 <https://yarnpkg.com/en/docs/install>。
- (4) 安装 Bower，使用命令 `yarn global add bower`。
- (5) 安装 Gulp，使用命令 `yarn global add gulp-cli`。
- (6) 如果想使用 JHipster 应用市场，就需要安装 Yeoman，执行命令 `yarn global add yo`。
- (7) 最后安装 JHipster，使用命令 `yarn global add generator-jhipster`。

本地安装使用 NPM:

- (1) 安装 JDK 8。
- (2) 安装 Node.js，通过网址 <https://nodejs.org/en/> 安装 LTS 版本。
- (3) 使用命令更新 NPM，执行命令 `npm install -g npm`。
- (4) 使用 `npm install -g` 代替 `yarn global add`，其他步骤与 Yarn 安装方式相同。

➤ Docker 方式安装

推荐使用 Docker 的方式安装，可以直接使用 Dockerhub 上的镜像，如果是 Windows 平台，则可以使用 Docker Toolbox。

使用命令下载最新的 master 版本：

```
docker image pull jhipster/jhipster:master
```

所有版本的 tags 可以从 <https://hub.docker.com/r/jhipster/jhipster/tags/>上看到，也可以直接使用 Dockerfile 进行构建。

8.3.2 使用

安装完成后，就可以使用 JHipster 来构建工程了。

在当前用户的根目录下，新建文件夹 jhipster，使用命令：

```
mkdir ~/jhipster
```

然后运行启动 Docker，确保 8080、9000 和 3001 端口没有被占用。如果被占用，则需要启动时将 Docker 的端口映射修改为其他未被占用的端口。

使用 Docker run 运行镜像，使用命令：

```
docker container run --name jhipster -v ~/jhipster:/home/jhipster/app -v ~/m2:/home/jhipster/.m2 -p 8090:8080 -p 9000:9000 -p 3001:3001 -d -t jhipster/jhipster
```

因为笔者本地的 8080 端口被占用，所以在命令中使用 8090 来做端口映射。

然后检查容器是否已经运行，使用命令：

```
docker container ps
```

得到如下信息：

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS		NAMES
2eb7ef8b2082	jhipster/jhipster	"tail -f /home/jhi..."	About a minute ago
Up	About a minute	0.0.0.0:3001->3001/tcp, 0.0.0.0:9000->9000/tcp, 0.0.0.0:8090->8080/tcp	jhipster

表示容器已经正常运行，并且内部和外部的端口映射关系已经生成。

如果想停止运行，则使用 `docker container stop` 命令，重新开启可使用 `docker container start` 命令。

容器成功运行后，进入容器，关于进入容器的方式，在 Docker 相关的章节中已经有相关说明，这里使用 `docker container exec` 的方式进入容器，执行如下命令：

```
docker container exec -it --user root jhipster bash
```

进入容器后，就可以构建工程，因为容器中已经将上面所有的依赖都安装完成了。

输入 `jhipster`，可以看到如图 8-5 所示的界面。

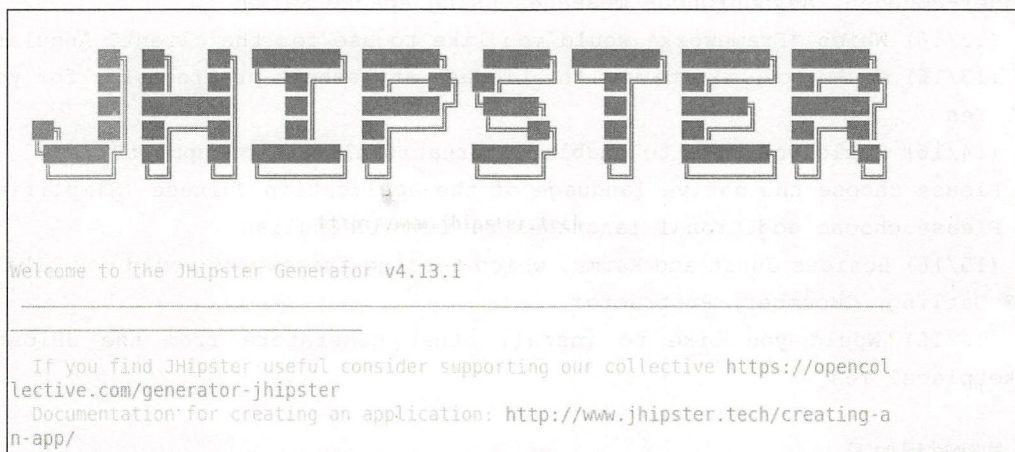


图 8-5 启动 JHipster

8.3.3 构建单体应用

先从单体应用开始，使用 JHipster 生成整个工程结构：

- (1/16) Which **type** of application would you like to create? Monolithic application (recommended for simple projects)
- (2/16) What is the base name of your application? single
- (3/16) What is your default Java package name? com.cloud.sky
- (4/16) Do you want to use the JHipster Registry to configure, monitor and scale your application? No
- (5/16) Which **type** of authentication would you like to use? JWT

authentication (stateless, with a token)

(6/16) Which *type* of database would you like to use? SQL (H2, MySQL, MariaDB, PostgreSQL, Oracle, MSSQL)

(7/16) Which *production* database would you like to use? MySQL

(8/16) Which *development* database would you like to use? H2 with disk-based persistence

(9/16) Do you want to use Hibernate 2nd level cache? No

(10/16) Would you like to use Maven or Gradle for building the backend? Maven

(11/16) Which other technologies would you like to use? Social login (Google, Facebook, Twitter), Search engine using Elasticsearch, Clustered HTTP sessions using Hazelc

ast, WebSockets using Spring Websocket, API first development using swagger-codegen, Asynchronous messages using Apache Kafka

(12/16) Which *Framework* would you like to use for the client? Angular 4

(13/16) Would you like to use the LibSass stylesheet preprocessor for your CSS? Yes

(14/16) Would you like to enable internationalization support? Yes

Please choose the native language of the application Chinese (Simplified)

Please choose additional languages to install English

(15/16) Besides Junit and Karma, which testing frameworks would you like to use? Gatling, Cucumber, Protractor

(16/16) Would you like to install other generators from the JHipster Marketplace? Yes

构建过程如下。

- (1) 选择构建单体应用，选择 Monolithic application (recommended for simple projects)。
- (2) 输入工程名，工程名称可以根据自身需要设定。
- (3) 输入包名，也是根据自身需要输入的。
- (4) 选择是否使用注册中心，因为是单体应用，可选择不使用注册中心。
- (5) 确定认证方式，选择 JWT authentication (stateless, with a token) 的认证方式。
- (6) 选择使用的数据库类型，一般的应用中都使用关系型数据库比较多，作为单体应用，选择 SQL (H2、MySQL、MariaDB、PostgreSQL、Oracle、MSSQL)。
- (7) 选择生产环境要使用的数据库，选择 MySQL。
- (8) 选择开发使用的数据库，选择 H2 with disk-based persistence。
- (9) 询问是否使用 Hibernate 二级缓存，可以根据项目的需要选择，因为是测试应用，这

里选择不使用缓存。

(10) 选择使用的构建工具，可以根据自己的熟悉程度进行选择，默认选择 Maven。

(11) 选择是否还使用其他框架，比如 OAuth 2.0、WebSocket 等，不使用就选 NO。

(12) 选择前端的框架，可选的有 AngularJS 1 版本和 Angular 4 版本。可以根据自身对框架的熟悉程度进行选择。

(13) 选择是否使用 LibSass 处理 CSS，也可以根据对框架的熟悉程度进行选择。

(14) 选择是否支持国际化，一般的项目中对国际化的要求还是比较高的，所以选择 yes。

(15) 选择国际化支持的语言，一般选择简单中文和英语。

(16) 选择是否使用 JHipster 市场的应用，如果不需要就选择 NO。

经过以上的步骤，应用构建完成。

然后需要构建实体。

实体是应用中的基本对象。创建实体的步骤如下：

(1) 在 DOS 窗口进入工程所在目录，运行命令 `yo jhipster:entity author`，回车。

(2) 回答一系列的交互问题，目的是生成自己想要的 Author 对象，包括域、域对应的类型、是否验证、Author 是否与别的实体建立关系，等等。

实体的构建使用的是 JDL 语言，下面的章节重点讲解 JDL。

8.3.4 Entity sub-generator

JHipster 通过 `entity sub-generator` 自动创建前后端相应代码。

JHipster `entity sub-generator` 根据项目类型和选项自动创建相应代码(Gateway 和 Monolithic application 会创建前后端代码，uaa、microservice 创建后端代码)。

对于简单（字段少）、实体类数量少的实体创建，建议使用命令行。

`jhipster:entity` 可以自动生成代码。

`jhipster:entity` 的可选参数如下。

- `--table-name <table-name>`: 默认情况下 JHipster 会生成一个名为实体名字的表，如果想要一个不一样的表名，则可以在这里传递参数。
- `--angular-suffix <suffix>`: 如果想要所有的 Angular 路由都有一个自定义的后缀，则可以在这里传递参数。
- `--regenerate`: 重新生成一个已存在的实体。

- `--skip-server`: 跳过服务器端的代码, 只生成客户端的代码。
- `--skip-client`: 跳过客户端的代码, 只生成服务器端的代码。

对于比较复杂的实体(数量多、关系复杂、字段多), 建议使用 JHipster UML 和 JDL Studio。构建完应用, 就需要与数据库交互。JHipster 提供了一种定义语言, 称为 JDL。

JDL 是 JHipster 指定的领域语言, 用于构建实体之间的关系。它也是一个开源的项目, 地址是: <https://github.com/jhipster/jhipster-core>。

主要完成以下几件事:

- 声明实体及其属性;
- 声明实体之间的关系;
- 声明一些 JHipster 特定的选项。

可以通过在线可视化的方式对 JDL 进行编辑, 地址是: <https://start.jhipster.tech/jdl-studio/>。也可以使用 `import-jdl` 生成想要的实体。使用命令:

```
yo jhipster:import-jdl your-jdl-file.jh
```

➤ 实体声明

```
entity <entity name> {  
  <field name> <type> [<validation>*]  
}
```

实体声明的语法如下:

- `<entity name>`是实体的名字;
- `<field name>`是实体的字段属性的名字;
- `<type>`是 JHipster 支持的字段属性;
- `<validation>`是可选选项, 字段是否进行校验。

支持的数据类型及对应关系如表 8-1 所示。

表 8-1 JDL 数据类型及对应关系

SQL	MongoDB	Cassandra	Validations
String	String	String	required, minlength, maxlength, pattern
Integer	Integer	Integer	required, min, max
Long	Long	Long	required, min, max

续表

SQL	MongoDB	Cassandra	Validations
BigDecimal	BigDecimal	BigDecimal	required, min, max
Float	Float	Float	required, min, max
Double	Double	Double	required, min, max
Enum	Enum		required
Boolean	Boolean	Boolean	required
LocalDate	LocalDate		required
		Date	required
ZonedDateTime	ZonedDateTime		required
		UUID	required
Blob	Blob		required, minbytes, maxbytes
AnyBlob	AnyBlob		required, minbytes, maxbytes
ImageBlob	ImageBlob		required, minbytes, maxbytes
TextBlob	TextBlob		required, minbytes, maxbytes
Instant	Instant	Instant	required

JDL 的设计理念是简单使用和简单可读，直接通过面向对象的方式处理业务。

注意：JHipster 会增加一个默认 id 字段，所以不用单独处理这个字段。

➤ 关系声明

```
relationship (OneToMany | ManyToOne | OneToOne | ManyToMany) {
  <from entity>[<relationship name>] to <to entity>[<relationship
name>]]
}
```

关系声明语法如下。

- (OneToMany | ManyToOne | OneToOne | ManyToMany)是关系的类型。
- <from entity>是拥有此关系的实体拥有者：源。
- <to entity>是这个关系连接到的实体名字：目标。
- <relationship name>实体关系的别名和类型。
- required: 注入的字段是否是必需的。

下面是一个简单的例子。

一本书必须有一个作者，一个作者可能有几本书，所以生成的实体关系代码如下：

```
entity Book
entity Author

relationship OneToMany {
    Author{book} to Book{writer(name) required}
}
```

➤ 枚举

声明枚举，用于类型标识，比如：

```
enum Language {
    CHINESE, ENGLISH, SPANISH
}
```

然后可以直接在实体中直接使用声明的枚举：

```
entity Book {
    title String required,
    description String,
    language Language
}
```

8.3.5 开发和运行

为了开发和演示方便，工程以实体文件都使用 JHipster Online 的方式生成。

生成的项目地址为 <https://github.com/cloudskyme/single>。

二次开发依赖的环境必须安装 Node.js，安装 Yarn。

打开 STS，选择“File”→“import”，找到 Existing Maven Projects，然后找到生成的工程源码并导入。

使用管理员身份打开控制台，进入工程目录，运行 `yarn install`。

然后在工程中运行 `SingleApp.java` 主函数。运行成功后可以得到如下信息：

```
2017-12-25 14:45:26.619 INFO 12684 --- [ restartedMain]
com.cloud.skyme.SingleApp : Started SingleApp in 16.589 seconds
(JVM running for 18.553)
2017-12-25 14:45:26.620 INFO 12684 --- [ restartedMain]
```



```
com.cloud.skyme.SingleApp :
-----
Application 'single' is running! Access URLs:
Local:           http://localhost:8080
External:        http://192.168.120.9:8080
Profile(s):      [swagger, dev]
```

在 yarn install 完成后, 运行 yarn start。

运行成功后, 系统会自动弹出默认浏览器, 并且打开默认页面。

右上角分别是首页链接、语言切换, 如果生成的过程中选择国际化, 这里会显示国际化的各种语言、账号登录和注册, 如图 8-6 所示。



图 8-6 单体应用操作界面

单击登录, 默认的管理用户的用户名和密码都是 admin。登录成功后, 可以看到多出数据和管理两个菜单。数据部分是根据 JDL 生成的实体完成基础的 CRUD 操作。管理部分如下。

➤ 用户管理

管理系统用户与相应的角色信息, 可以在这里维护系统用户的相关信息。

➤ 资源监控

查看系统资源使用情况、HTTP 请求情况、服务被调用情况、缓存情况及数据源使用情况。

➤ 服务状态

查看服务器的硬盘使用情况和数据使用情况。

➤ 配置

查看所有资源的配置信息。

➤ 审核

查看应用的审核情况。

➤ 日志

设置所有包的日志级别，可以根据需要动态修改。

➤ API

基于 Swagger 的控制层的接口参数。

➤ 数据库

连接工程中使用的 H2 数据库，可以在此对测试数据进行处理。

8.3.6 插件安装

所谓“工欲善其事，必先利其器”，可见一个好用的工具对开发起着什么样的作用。

在 STS 中的安装方式：Help→Eclipse Marketplace。

输入 JHipster，单击安装，然后单击向下，直到安装完成。

8.4 目录结构

前端的主要代码都在 `src/main/webapp` 下，未建实体时，目录结构如下：

```
webapp
├── app - Your application (你的应用)
│   ├── account - User account managementUI (用户账号管理界面)
│   ├── admin - Administration UI (管理员界面)
│   └── blocks - Common building blocks like configuration and interceptors
│       (公共构建模块和拦截器)
│   ├── components - Common components like alerting and form validation (常
│       用组件，比如警告组件和验证组件)
│   ├── entities - Generated entities (more information below) (生成的实体)
│   ├── home - Home page (主页)
│   └── layouts - Common page layouts like navigation bar and error pages
│       (通用页面布局，类似导航条和错误页)
└── services - Common services like authentication and user management
```


(通用服务, 类似身份验证和管理)

- | |— app.constants.js - Application constants (应用常量)
- | |— app.module.js - Application modules configuration (应用 modules 配置)
- | |— app.state.js - Main application router (应用路由--单页应用业务通过 JS 控制, 无法简单通过 URL 控制, 故而使用应用路由)
- |— bower_components - Dependencies installed by Bower (通过 Bower 安装的依赖)
- |— content - Static content (静态内容)
- | |— images - Images (图片)
- | |— styles - CSS stylesheets (css 样式表)
- | |— fonts - Font files will be copied here (字体库)
- |— i18n - Translation files (国际化语言文件)
- |— scss - Sass style sheet files will be here if you choose the option

(如果创建应用选择了 LibSass, 则其文件会在这里生成)

- |— swagger-ui - Swagger UI front-end (SwaggerAPI 文档前段页面)
- |— 404.html - 404 page (404 错误页)
- |— favicon.ico - Fav icon (网站图标)
- |— index.html - Index page (索引页)
- |— robots.txt - Configuration for bots and Web crawlers (针对搜索引擎爬虫的配置文件)

当创建实体 Author 后, 在 script 下多了一个 entities:

后台主要的 CRUD 操作代码在 com. Your application.testjhipster.web.rest 包下。

8.5 构建微服务应用

JHipster 生成的微服务主要分为: 网关、Traefik、注册中心、Consul、JHipster UAA、微服务应用和 ELK。

根据项目类型, 会自动创建相应的代码, 网关和单体应用会创建前后端代码, uaa、microservice 创建后端代码。

8.5.1 注册中心

JHipster registry 是一个基于 Spring Cloud 的配置中心。所有的微服务应用都需要注册到这里。它是一个 Eureka Server, 这个服务能发现所有的应用, 并且实现路由、负责均衡等功能。同时它也是一个配置中心, 为所有的服务提供运行时的配置信息。它也是一个管理服务, 为所有的服务提供监控面板。

这个项目是一个开源项目，默认不需要进行改动，开源网址为 <https://github.com/jhipster/jhipster-registry>。

JHipster Registry 有两种模式，一种是 dev 模式，一种是 prod 模式，分别对应的就是开发模式和生产模式。

开发模式加载本地的配置文件使用 dev 模式，从工程的 central-config 中加载配置文件。

下载源代码，并且编译运行。在命令行执行如下命令：

```
mvnw -Pdev
```

启动后运行，打开一个命令行工具，输入如下命令：

```
Yarn & Yarn start
```

自动弹出默认浏览器，并且自动输入地址 <http://localhost:9000/#/>。

使用默认的用户名 admin，密码也是 admin。登录后可以看到 Eureka、Configuration 及管理控制台的菜单。

登录之后的界面如图 8-7 所示。

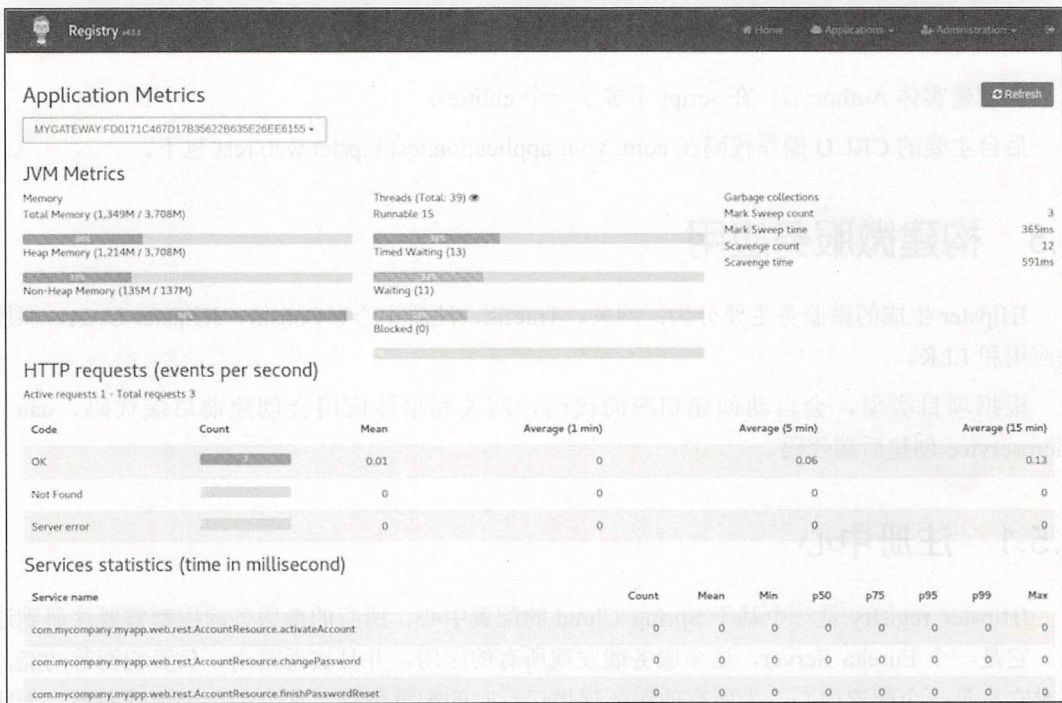


图 8-7 JHipster 注册中心

生产模式参数使用 prod。

如果想打包，则需要执行如下命令：

```
./mvnw -Pprod
```

也可以使用官方发布的 release 版本，下载地址是：

<https://github.com/jhipster/jhipster-registry/releases>。

8.5.2 创建微服务网关

JHipster 可以生成 API 网关，网关其实就是一个普通的 JHipster 应用，作为微服务的入口，为所有的微服务提供 HTTP 路由、负载均衡、安全和 API 文档的功能。

JHipster 网关的架构如图 8-8 所示。

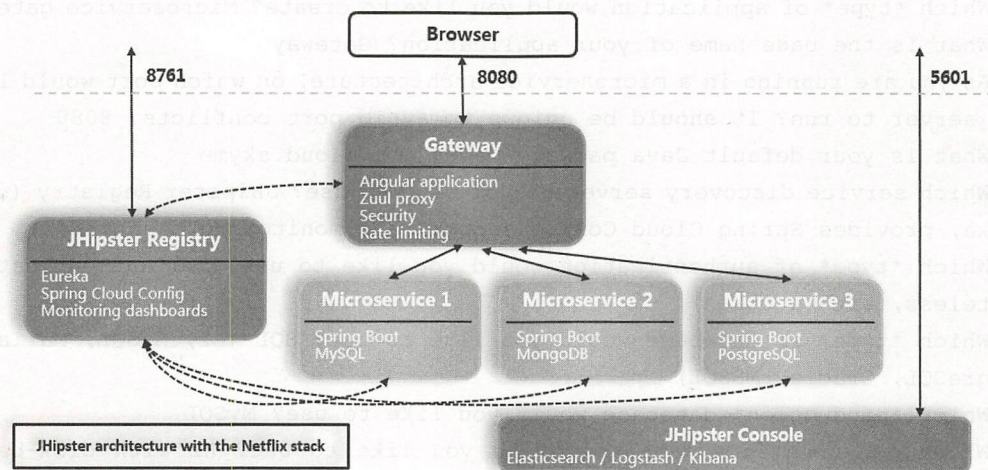


图 8-8 JHipster 网关架构

当网关和微服务运行后，就可以在注册中心中看到相应的服务，使用的是 `src/main/resources/config/application.yml` 中的 `eureka.client.serviceUrl.defaultZone` 配置。网关将自动代理所有的到微服务的请求。网关的路由功能使用的是 Spring Cloud 中的 Eureka，当把一个微服务应用 App 注册到 Eureka 后，就可以通过网关配置的代理访问到/app 微服务应用。负载均衡使用的是 Spring Cloud 中的组件 Ribbon，熔断器使用的是 Spring Cloud 中的组件 Hystrix。

网关为所有请求提供安全验证，可以基于 JWT、OpenID，也可以基于 JHipster UAA。

自动文档使用的是 Swagger API，在管理菜单中有一项是所有 API 列表。

网关也提供了限流的功能，可以使用匿名用户的 IP 进行限流，也可以针对登录用户的访问进行限流。限流是否开启是在配置文件 `application-dev.yml` 或 `application-prod.yml` 中设置的。设置内容如下：

```
jhipster:
  gateway:
    rate-limiting:
      enabled: true
```

构建应用，在命令行输入：

Jhipster

会提示输入交互：

```
Which *type* of application would you like to create? Microservice gateway
What is the base name of your application? Gateway
As you are running in a microservice architecture, on which port would like
your server to run? It should be unique to avoid port conflicts. 8080
What is your default Java package name? com.cloud.skyme
Which service discovery server do you want to use? JHipster Registry (uses
Eureka, provides Spring Cloud Config support and monitoring dashboards)
Which *type* of authentication would you like to use? JWT authentication
(stateless, with a token)
Which *type* of database would you like to use? SQL (H2, MySQL, MariaDB,
PostgreSQL, Oracle, MSSQL)
Which *production* database would you like to use? MySQL
Which *development* database would you like to use? H2 with disk-based
persistence
Do you want to use Hibernate 2nd level cache? No
Would you like to use Maven or Gradle for building the backend? Maven
Which other technologies would you like to use? Search engine using
Elasticsearch, Clustered HTTP
sessions using Hazelcast, WebSockets using Spring Websocket, API first
development using swagger-c
odegen, Asynchronous messages using Apache Kafka
Which *Framework* would you like to use for the client? Angular 5
Would you like to enable *SASS* support using the LibSass stylesheet
preprocessor? No
```



```
Would you like to enable internationalization support? Yes
Please choose the native language of the application English
Please choose additional languages to install Chinese (Simplified)
Besides Junit and Karma, which testing frameworks would you like to use?
Gatling, Cucumber, Protractor
Would you like to install other generators from the JHipster Marketplace? No
```

构建过程如下。

- (1) 选择构建微服务网关应用。选择 `MicroService Gateway`。
- (2) 输入工程名，工程名称可以根据自身需要设定。
- (3) 询问要使用的安全验证端口，默认是 8080，如不修改可选择 `yes`。
- (4) 输入包名，也是根据自身需要输入的。
- (5) 选择是否使用注册中心，因为是微服务应用，选择使用注册中心 `JHipster Registry (uses Eureka, provides Spring Cloud Config support and monitoring dashboards)`。
- (6) 确定认证方式，选择 `JWT authentication (stateless, with a token)` 的认证方式。
- (7) 选择使用的数据库类型，一般的应用中都使用关系型数据库比较多，作为单体应用，选择 `SQL (H2、MySQL、MariaDB、PostgreSQL、Oracle、MSSQL)`。
- (8) 选择生产环境要使用的数据库，选择 `MySQL`。
- (9) 选择开发使用的数据库，选择 `H2 with disk-based persistence`。
- (10) 询问是否使用 `Hibernate` 二级缓存，可以根据项目的需要选择。
- (11) 选择使用的构建工具，可以根据自己的熟悉程度进行选择，默认是选择 `Maven`。
- (12) 选择是否还使用其他框架，比如 `OAuth 2.0`、`WebSocket` 等，不使用就选 `no`。
- (13) 选择前端的框架，可选的有 `AngularJS 1` 版本和 `Angular 4` 版本。可以根据自身对框架的熟悉程度进行选择。
- (14) 选择是否使用 `LibSass` 处理 `CSS`，这个也可以根据对框架的熟悉程度进行选择。
- (15) 选择是否支持国际化，一般的项目中对国际化的要求还是比较高的，所以选择 `yes`。
- (16) 选择国际化支持的语言，一般选择简单中文和英语。
- (17) 选择是否使用 `JHipster` 市场的应用，如果不需要就选择 `no`。

经过以上的步骤，应用构建完成。

这里需要注意的一点是，微服务的应用默认会使用消息队列进行属性变更的通知，所以在运行应用前，需要在本机安装 `ZooKeeper` 和 `Kafka`，微服务的应用默认会使用 `Spring Cloud Stream`

进行消息驱动的开发。

进入网关目录，使用开发环境，执行如下命令：

```
mvnw -Pdev
```

运行出现如下内容代表输入成功：

```
-----
Application 'gateway' is running! Access URLs:
Local:           http://localhost:8080
External:        http://192.168.198.138:8080
Profile(s):      [swagger, dev]
-----

2018-01-27 22:42:54.112 INFO 11908 --- [ restartedMain]
com.cloud.skyme.GatewayApp           :
-----

Config Server:      Connected to the JHipster Registry config server!
-----
```

启动成功后，在注册中心中能够看到网关的微服务应用。

8.5.3 Traefik

Traefik 是一种 HTTP 反向代理和负载均衡器，可轻松部署微服务。它可以像 Zuul 那样路由 HTTP 请求，因此它与 JHipster 网关的功能相似，但是它比 API 网关的工作级别更低：它仅路由 HTTP 请求，不提供速率限制、安全和 Swagger 文档聚合。

Traefik 的开源地址为 <https://github.com/containous/traefik>。

它可以单独使用，也可以把路由放到网关之前。有一点是需要注意的，它只与 Consul 一起工作，这也就意味着如果使用它，就不能使用 JHipster Registry。

8.5.4 JHipster UAA

JHipster UAA 是一个使用 OAuth 2 授权协议保护 JHipster 微服务的用户验证和授权服务。它被设计成一个单独的应用，用于安全验证。

JHipster UAA 的应用架构如图 8-9 所示。

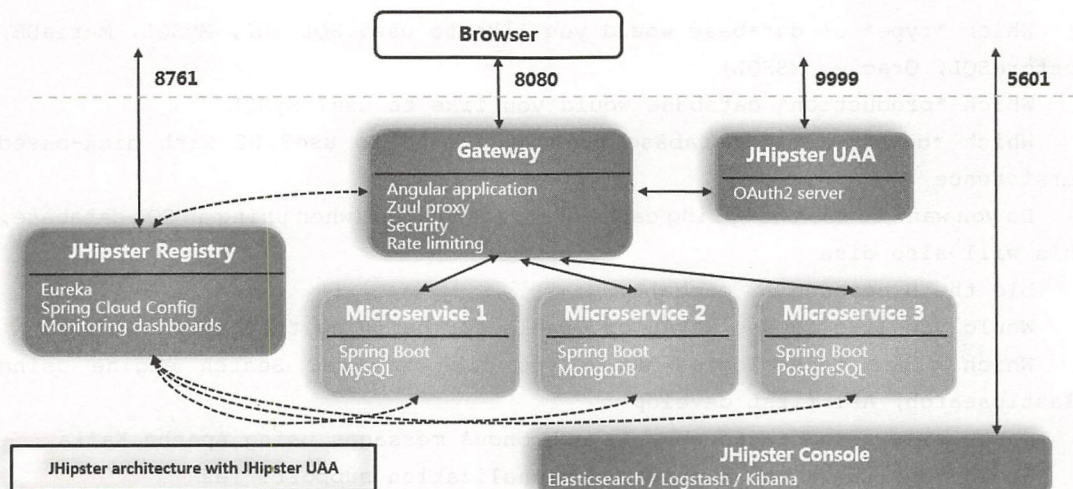


图 8-9 JHipster UAA 的应用架构图

基本安装要求:

- JHipster UAA 服务;
- 至少一个微服务 (使用 UAA 认证);
- 一个 JHipster gateway 网关 (使用 UAA 认证)。

构建应用, 在命令行输入:

```
Jhipster
```

会提示输入交互:

```
Application files will be generated in folder: /root/micro
Which *type* of application would you like to create? JHipster UAA server
(for microservice OAuth2 authentication)
What is the base name of your application? uaa
As you are running in a microservice architecture, on which port would like
your server to run? It should be unique to avoid port conflicts. 9999
What is your default Java package name? com.cloud.skyme
Which service discovery server do you want to use? JHipster Registry (uses
Eureka, provides Spring Cloud Config support and monitoring dashboards)
```

```

Which *type* of database would you like to use? SQL (H2, MySQL, MariaDB,
PostgreSQL, Oracle, MSSQL)
Which *production* database would you like to use? MySQL
Which *development* database would you like to use? H2 with disk-based
persistence
Do you want to use the Spring cache abstraction? No (when using an SQL database,
this will also disa
ble the Hibernate L2 cache)
Would you like to use Maven or Gradle for building the backend? Maven
Which other technologies would you like to use? Search engine using
Elasticsearch, API first develop
ment using swagger-codegen, Asynchronous messages using Apache Kafka
Would you like to enable internationalization support? Yes
Please choose the native language of the application English
Please choose additional languages to install Chinese (Simplified)
Besides Junit and Karma, which testing frameworks would you like to use?
Gatling, Cucumber
Would you like to install other generators from the JHipster Marketplace?
No

```

构建过程如下。

- (1) 选择构建 UAA 应用，使用 OAuth 2。选择 JHipster UAA server (for microservice OAuth2 authentication)。
- (2) 输入工程名，工程名称可以根据自身需要设定。
- (3) 询问要使用的安全验证端口，默认是 9999，如不修改可选择 yes。
- (4) 输入包名，也是根据自身需要输入的。
- (5) 选择是否使用注册中心，因为是微服务应用，选择使用注册中心 JHipster Registry (uses Eureka, provides Spring Cloud Config support and monitoring dashboards)。
- (6) 选择使用的数据库类型，一般的应用中都使用关系型数据库比较多，作为单体应用，选择 SQL (H2、MySQL、MariaDB、PostgreSQL、Oracle、MSSQL)。
- (7) 选择你生产环境要使用的数据库，选择 MySQL。
- (8) 选择开发使用的数据库。选择 H2 with disk-based persistence。
- (9) 询问是否使用 Hibernate 二级缓存，可以根据项目的需要选择。
- (10) 选择使用的构建工具，可以根据自己的熟悉程度进行选择，默认是选择 Maven。

(11) 选择是否还使用其他框架，比如 OAuth 2.0、WebSocket 等，不使用就选 no。

(12) 选择是否支持国际化，一般项目中对国际化的要求还是比较高的，所以选择 yes。

(13) 选择国际化支持的语言，一般选择简单中文和英语。

(14) 选择是否使用 JHipster 市场的应用，如果不需要就选择 no。

经过以上的步骤，应用构建完成。

8.5.5 构建微服务应用

先从单体应用开始，使用 JHipster 生成整个工程结构：

(1/16) Which **type** of application would you like to create? Microservice application

(2/16) What is the base name of your application? Microservice

(3/16) As you are running in a microservice architecture, on which port would like your server to run? It should be unique to avoid port conflicts. 8081

(4/16) What is your default Java package name? com.cloud.sky

(5/16) Do you want to use the JHipster Registry to configure, monitor and scale your application? JHipster Registry (uses Eureka, provides Spring Cloud Config support and monitoring dashboards)

(6/16) Which **type** of authentication would you like to use? JWT authentication (stateless, with a token)

(7/16) Which **type** of database would you like to use? SQL (H2, MySQL, MariaDB, PostgreSQL, Oracle, MSSQL)

(8/16) Which **production** database would you like to use? MySQL

(9/16) Which **development** database would you like to use? H2 with disk-based persistence

(10/16) Do you want to use Hibernate 2nd level cache? No

(11/16) Would you like to use Maven or Gradle for building the backend? Maven

(12/16) Which other technologies would you like to use? Social login (Google, Facebook, Twitter), Search engine using Elasticsearch, Clustered HTTP sessions using Hazelcast, WebSockets using Spring WebSocket, API first development using swagger-codegen, Asynchronous messages using Apache Kafka

(13/16) Which **Framework** would you like to use for the client? Angular 4

(14/16) Would you like to use the LibSass stylesheet preprocessor for your CSS? Yes

(15/16) Would you like to enable internationalization support? Yes

```
Please choose the native language of the application Chinese (Simplified)
Please choose additional languages to install English
(16/16) Besides Junit and Karma, which testing frameworks would you like to
use? Gatling, Cucumber, Protractor
(17/16) Would you like to install other generators from the JHipster
Marketplace? Yes
```

构建过程如下。

- (1) 选择构建微服务应用，选择 `Microservice application`。
- (2) 要求输入工程名，工程名称可以根据自身需要设定。
- (3) 询问创建微服务所使用的端口，默认是 8081，根据自身服务的情况可以选择不同的端口。
- (4) 输入包名，也是根据自身需要输入的。
- (5) 选择是否使用注册中心，因为是微服务应用，需要选择使用注册中心 `JHipster Registry (uses Eureka, provides Spring Cloud Config support and monitoring dashboards)`。
- (6) 确定认证方式，选择 `JWT authentication (stateless, with a token)` 的认证方式。
- (7) 选择使用的数据库类型，一般的应用中都使用关系型数据库比较多，作为单体应用，选择 `SQL (H2、MySQL、MariaDB、PostgreSQL、Oracle、MSSQL)`。
- (8) 选择生产环境要使用的数据库，选择 `MySQL`。
- (9) 选择开发使用的数据库，选择 `H2 with disk-based persistence`。
- (10) 询问是否使用 `Hibernate` 二级缓存，可以根据项目的需要选择，因为是测试应用，这里选择不使用缓存。
- (11) 选择使用的构建工具，可以根据自己的熟悉程度进行选择，默认是选择 `Maven`。
- (12) 选择是否还使用其他框架，比如 `OAuth 2.0`、`WebSocket` 等，不使用就选 `no`。
- (13) 选择前端的框架，可选的有 `AngularJS 1` 版本和 `Angular 4` 版本。可以根据自身对框架的熟悉程度进行选择。
- (14) 选择是否使用 `LibSass` 处理 `CSS`，这个也可以根据对框架的熟悉程度进行选择。
- (15) 选择是否支持国际化，一般的项目中对国际化的要求还是比较高的，所以选择 `yes`。
- (16) 选择国际化支持的语言，一般选择简单中文和英语。
- (17) 选择是否使用 `JHipster` 市场的应用，如果不需要就选择 `no`。

经过以上的步骤，应用构建完成。

使用开发环境，执行如下命令：

```
mvnw -Pdev
```

启动构建的应用：

```
-----
Application 'micro1' is running! Access URLs:
Local:      http://localhost:8081
External:   http://192.168.198.138:8081
Profile(s): [swagger, dev]
-----
2018-01-27 23:44:54.264 INFO 12795 --- [ restartedMain]
com.cloud.skyme.Micro1App      :
-----
Config Server:      Connected to the JHipster Registry config server!
-----
```

启动完成后，在注册中心能够看到启动的服务。

8.6 基础配置

JHipster 有两种 Spring 配置文件。

- dev 开发环境配置：为了开发的方便，可以直接连接开发环境或者测试环境。
- prod 生产环境配置：一般连接生产环境，需要考虑性能和可扩展性。

8.6.1 JHipster 属性配置

JHipster 提供了特定的应用属性，这些属性是所有 JHipster 项目中的标准，这些属性使用 `io.github.jhipster.config.JHipsterProperties` 类来配置。

```
jhipster:
  # 用于 JHipster 异步函数调用的线程池
  async:
    core-pool-size: 2 # 初始化池大小
    max-pool-size: 50 # 最大池大小
    queue-capacity: 10000 # 池队列容量
```

```

# HTTP 配置
http:
    # V_1_1 for HTTP/1.1 or V_2_0 for HTTP/2.
    # 使用 HTTP/2 需要 SSL 支持(见 Spring Boot "server.ssl" 配置)
    version: V_1_1
    cache: # 用于 io.github.jhipster.web.filter.CachingHttpHeadersFilter
        timeToLiveInDays: 1461 # 静态内容默认缓存 4 年
# Hibernate 二级缓存, 用于 CacheConfiguration
cache:
    hazelcast: # Hazelcast configuration
        time-to-live-seconds: 3600 # 默认对象在缓存中保持 1 小时
        backup-count: 1 # 对象备份数量
    ehcache: # Ehcache 配置
        time-to-live-seconds: 3600 # 默认对象在缓存中保持 1 小时
        max-entries: 100 # 每次缓存开启时对象的最大数量
# E-mail 属性
mail:
    from: jhipster@localhost # 默认的 E-mail 发出地址
    base-url: http://127.0.0.1:8080 # 在邮件中使用的应用的 URL
# Spring 安全相关配置
security:
    remember-me: # JHipster 对“记住我”机制的安全实现: 基于会话的身份验证
        # 安全 key (对于你的应用是独有的且应保密)
        key: 0b32a651e6a65d5731e869dc136fb301b0a8c0e4
    client-authorization: # 用于 JHipster UAA 验证
        access-token-uri: # JHipster UAA 服务器 OAuth 令牌的 URL
        token-service-id: # 当前应用的 ID
        client-id: # OAuth 客户 ID
        client-secret: # OAuth 客户秘密
    authentication:
        jwt: # JHipster 指定 JWT 实现
            secret: # JWT 密钥
            token-validity-in-seconds: 86400 # 令牌在 24 小时内有效
            token-validity-in-seconds-for-remember-me: 2592000
            # “记住我”令牌在 30 天内有效
        oauth: # 用于 JHipster OAuth 2 对于 MongoDB 的特定实现
            client-id: # OAuth 客户 ID
            client-secret: # OAuth 客户秘密

```



```

        token-validity-in-seconds: 1800 # 令牌在 30 分钟内有效
# DropWizard Metrics 配置, 用于 MetricsConfiguration
metrics:
    jmx: # 作为 JMX beans 导出指标
        enabled: true # JMX 默认开启
    # 将指标发送给 Graphite 服务器
    # 使用 "graphite" Maven 配置文件以产生 Graphite 依赖
    graphite:
        enabled: false # Graphite 默认关闭
        host: localhost
        port: 2003
        prefix: jhipster
    # 将指标发送给 Prometheus 服务器
    # 使用 "Prometheus" Maven 配置文件以产生 Prometheus 依赖
    prometheus:
        enabled: false # Prometheus 默认关闭
        endpoint: /prometheusMetrics
    logs: # 在日志中报告 Dropwizard 指标
        enabled: false
        reportFrequency: 60 # 每秒报告的频度
# Logging 配置, 用于 LoggingConfiguration
logging:
    logstash: # 通过 socket 将日志提交给 Logstash
        enabled: false # Logstash 默认关闭
        host: localhost # Logstash 服务器 URL
        port: 5000 # Logstash 服务器端口
        queue-size: 512 # 缓存日志的队列
    spectator-metrics: # 在日志中报告 Netflix Spectator 指标
        enabled: false # Spectator 默认关闭
# Spring Social 对 Twitter/Facebook/Google 验证的特定配置
social:
    redirect-after-sign-in: "/#/home" # 验证成功后的重定向 URL
# cross-origin resource sharing (CORS) 默认关闭, 去掉注释以开启
# 配置标准的 org.springframework.web.cors.CorsConfiguration
cors:
    allowed-origins: "*"
    allowed-methods: GET, PUT, POST, DELETE, OPTIONS
    allowed-headers: "*"

```

```
exposed-headers:
  allow-credentials: true
  max-age: 1800
# JHipster 应用首页左上角的丝带显示
ribbon:
  #逗号分隔显示丝带的配置文件列表
  display-on-active-profiles: dev
```

8.6.2 作为 Maven 项目

可以通过 Maven 来启动 Java 服务器, JHipster 提供 Maven Wrapper, 所以不需要安装 Maven 并且能保证所有的项目使用者有相同的 Maven 版本。

Mvnw 会运行默认的 Maven 任务: Spring-boot:run。

应用就可以通过 <http://localhost:8080> 访问。

如果安装了指定的 Maven 版本, 就可以使用 Maven 的 run 命令启动 Java 服务器。

8.6.3 数据库

数据库可以使用 MySQL、MariaDB、PostgreSQL、MSSQL、MongoDB 或 Cassandra。

相应地在 `src/main/resources/config/application-*/yaml` 中配置 Spring Boot 的属性。

开发中一般建议使用 H2 数据库, 可以通过默认的 <http://localhost:8080/h2-console> 来访问它的控制台。

为了连接上数据库, 选择预配置的选项:

```
Driver Class: org.h2.Driver
JDBC URL: jdbc:h2:mem:jhipster
User name:
Password:
```

更新数据库

如果添加或修改了一个 JPA 实体, 需要更新数据库 schema。

JHipster 使用 Liquibase 来管理数据库更新, 将其存储到 `src/main/resources/config/liquibase/` 目录下的配置文件中。有三种方式可以使用 Liquibase: 使用实体生成器、使用 `liquibase:diff` Maven goal 或手动更新配置文件。

用实体生成器更新数据库

执行流程如下。

- 运行实体生成器。
- 一个新的“change log”会在 `src/main/resources/config/liquibase/changelog` 目录下创建，并且会自动加入 `src/main/resources/config/liquibase/master.xml` 文件中。
- 查看 change log，它会在下次运行应用时启用。

用 Maven 的 `liquibase:diff` 这个 goal 来更新数据库：

如果在开发中选择使用 MySQL、MariaDB、PostgreSQL，则可以使用 `./mvnw liquibase:diff` 来自动生成 changelog。

Liquibase Hibernate 是一个配置在 `pom.xml` 中的 Maven 插件，和 `Spring application.yml` 文件独立。所以如果改变了默认的设置，则需要同时改动这两份文件。

执行流程如下：

- 修改 JPA 实体。
- 编译应用（只对已编译的 Java 代码有效，所以不能忘记编译）。
- 运行 `./mvnw liquibase:diff`（或 `./mvnw compile liquibase:diff`）来编译之前的操作。
- 新的“change log”会在 `src/main/resources/config/liquibase/changelog` 目录下创建。

查看 change log 并将其加入 `src/main/resources/config/liquibase/master.xml` 文件中，这样在下次运行时它就会启用。

8.6.4 DTO

DTO 的使用原理如下。

当生成一个 JHipster 实体时，可以选择是否也生成 DTO，如果选择生成：

- 生成一个 DTO，映射到依赖的实体。
- 集合多对一的关系，只使用 ID 和用于展示的域。例如，一个 User 实体的多对一关系会将 `userId` 域和 `userLogin` 域添加到 DTO 中。
- 将忽略非所有者上的一对多、多对多关系，这符合实体的运作方式。
- 对于所有者的多对多关系，使用其他实体的 DTO 并将它们放在一个 Set 中使用，因此，只有当其他实体也使用 DTO 时才能支持。

使用 MapStruct 映射 DTO 和实体：

- DTO 看起来很像实体，经常需要将它们想办法相互映射。
- JHipster 选择的方式是使用 MapStruct，这是一个注解处理器，作为 Java 编译器的插件，将自动生成必要的映射。
- MapStruct 的 Mapper 被配置为 Spring Beans，并且支持独立注入。
- 建议将一个 Repository 注入一个 Mapper，这样就可以从 Mapper 中通过它的 ID 获取一个可控的 JPA 实体。

示例：

```
@Mapper
public abstract class CarMapper {

    @Inject
    private UserRepository userRepository;

    @Mapping(source = "user.id", target = "userId")
    @Mapping(source = "user.login", target = "userLogin")
    public abstract CarDTO carToCarDTO(Car car);

    @Mapping(source = "userId", target = "user")
    public abstract Car carDTOToCar(CarDTO carDTO);

    public User userFromId(Long id) {
        if (id == null) {
            return null;
        }
        return userRepository.findOne(id);
    }
}
```

8.6.5 分页

目前，如果使用 Cassandra 来创建应用，则是不能进行分页的。

分页使用了和 GitHub API 中一样的 Linker header。JHipster 提供对服务器和客户端特定的自定义实现。

生成实体时，JHipster 提供了 4 种分页选项：

- 不分页;
- 简单分页, 基于 Bootstrap pager;
- 完全分页系统, 基于 Bootstrap 分页组件;
- 有限滚动系统, 基于有限滚动指令。

8.6.6 文档

Swagger 是一个规范和完整的框架, 用于生成、描述、调用和可视化 RESTful 风格的 Web 服务。总体目标是使客户端和文件系统作为服务器以同样的速度来更新。文件的方法、参数和模型紧密集成到服务器端的代码, 允许 API 来始终保持同步。Swagger 让部署管理和使用功能强大的 API 变得简单。

src/main/resources/config/application.yml 下的 jhipster.swagger 如下:

```
jhipster:
  swagger:
    title: xxx API
    description: xxx API 文档
    version: 0.0.1
    termsOfServiceUrl:
    contactName: AnJia
    contactUrl:
    contactEmail: anjia0532@gmail.com
    license:
    licenseUrl:
```

主要配置文件有三处:

- com.xx.xx(默认包名下).config.apidoc.SwaggerConfiguration.java;
- com.xx.xx(默认包名下).config.apidoc.PageableParameterBuilderPlugin.java;
- com.xx.xx(默认包名下).config.JHipsterProperties.java 中的 Swagger 内部类。

Entity 注解配置

例如, Bar.java:

```
@ApiModel( value = "测试类")
public class Bar implements Serializable {}
```

```

@ApiModelProperty( required = true,example = "示例",allowableValues = "
男,女")
@Column(name = "sex")
private String sex;

```

optional 是可选值，如果设置 required=true，则不显示 optional，如果是枚举值，则通过 allowableValues="值 1,值 2"进行设置。

Controller 注解配置：

```

@Api
public class BarResource {}

```

@Api 中的 tags="department-resource",description="科室" 对应的是 swagger-ui 页面中的 department-resource:Department Resource:

```

@ApiOperation(value = "创建 Bar",
    notes = "创建 Bar")
@ApiResponses(value = {
    @ApiResponse(code = 404, message = "Bar 未找到") })
public ResponseEntity<Bar> createBar(@RequestBody Bar bar){}

```

国际化配置

Swagger 默认是英文：

```

<script src="../../bower_components/swagger-ui/dist/lang/translator.js"
type="text/javascript"></script>
<script src="../../bower_components/swagger-ui/dist/lang/zh-cn.js"
type="text/javascript"></script>

```

需要注意的是，zh-cn.js 相比 en.js 翻译的字段少，比如 Example Value、Click to set as parameter value 等。

如果需要用到国际化，需要自己修改../bower_components/swagger-ui/dist/lang/zh-cn.js:

```

'use strict';

/* jshint quotmark: double */
window.SwaggerTranslator.learn({

```


"Warning: Deprecated": "警告: 已过时",
"Implementation Notes": "实现备注",
"Response Class": "响应类",
"Status": "状态",
"Parameters": "参数",
"Parameter": "参数",
"Value": "值",
"Description": "描述",
"Parameter Type": "参数类型",
"Data Type": "数据类型",
"Response Messages": "响应消息",
"HTTP Status Code": "HTTP 状态码",
"Reason": "原因",
"Response Model": "响应模型",
"Request URL": "请求 URL",
"Response Body": "响应体",
"Response Code": "响应码",
"Response Headers": "响应头",
"Hide Response": "隐藏响应",
"Headers": "头",
"Try it out!": "试一下!",
"Show/Hide": "显示/隐藏",
"List Operations": "显示操作",
"Expand Operations": "展开操作",
"Raw": "原始",
"can't parse JSON. Raw result": "无法解析 JSON. 原始结果",
"Example Value": "示例",
"Click to set as parameter value": "点击设置参数",
"Model Schema": "模型架构",
"Model": "模型",
"apply": "应用",
"Username": "用户名",
"Password": "密码",
"Terms of service": "服务条款",
"Created by": "创建者",
"See more at": "查看更多:",
"Contact the developer": "联系开发者",
"api version": "API 版本",

```

"Response Content Type":"响应 Content Type",
"Parameter content type":"参数类型:",
"fetching resource":"正在获取资源",
"fetching resource list":"正在获取资源列表",
"Explore":"浏览",
"Show Swagger Petstore Example Apis":"显示 Swagger Petstore 示例 Apis",
"Can't read from server. It may not have the appropriate access-control-
origin settings.":"无法从服务器读取。可能没有正确设置 access-control-origin.",
"Please specify the protocol for":"请指定协议: ",
"Can't read swagger JSON from":"无法读取 swagger JSON",
"Finished Loading Resource Information. Rendering Swagger UI":"已加载
资源信息。正在渲染 Swagger UI",
"Unable to read api":"无法读取 api",
"from path":"从路径",
"server returned":"服务器返回"
});

```

另外有些单词，本身不含 data-sw-translate 属性，无法简单通过增加 zh-cn.js 的 k:v 进行翻译。

- 找到生成处，在其标签上增加 data-sw-translate 属性；
- 在生成处，直接将 value 改为翻译好的内容。

修改../bower_components/swagger-ui/dist/swagger-ui.js 文件。例如，修改 optional 为选填，大约为 5666 行，改前：

```

if(!propertyIsRequired) {
    html += ', <span class="propOptKey">optional</span>';
}

```

改后：

```

if(!propertyIsRequired) {
    html += ', <span class="propOptKey" data-sw-translate>optional</span>';
}

```

然后在 zh-cn.js 中增加：

```

"optional":"选填"

```


8.7 小结

JHipster 不仅仅是一套代码生成工具，它也是众多微服务领域新技术合集，学习和使用这套工具，能够让我们对业界领先的微服务的架构方式有清晰的认识，从中学习到更多的关于构建微服务所需的技能。



9 chapter

第 9 章

微服务之自动化部署



一般的公司对于自动化部署都持保守态度，认为自己的团队规模小，业务流程不够复杂，所以对于部署的自动化往往不够重视。我们一起来看一下在部署中常犯的错误：

- 手工部署软件；
- 开发完成后才提交部署；
- 生产环境全手工完成配置。

很多公司都使用手工的方式发布软件，在微服务中，持续集成和持续部署是最基础的工作之一。其中复杂的地方主要是对多种环境的构建支撑。项目中有用 Maven 编译的、有用 ANT 编译的，如果有移动应用，有 Android 系统的、iOS 系统的，还有一些前端应用的编译，比如 Node.js，这么多不同的环境我们怎么支持？另外，构建过程中还需要考虑和代码质量分析、单元测试、介质上传等能力的结合，这样的构建过程其实也是一个工作流程。

9.1 私有仓库搭建

私服是架设在局域网的一种特殊的远程仓库，目的是代理远程仓库及部署第三方构件。有了私服之后，当 Maven 需要下载构件时，直接请求私服，私服上存在则下载到本地仓库；否则，私服请求外部的远程仓库，将构件下载到私服，再提供给本地仓库下载。

9.1.1 Nexus 介绍

Nexus 为 Maven 仓库管理器。Nexus 提供了强大的仓库管理功能和构件搜索功能，它占用较少的内存，基于简单文件系统而非数据库。它是一套“开箱即用”的系统，不需要数据库，它使用文件系统加 Lucene 来组织数据。使用 ExtJS 来开发界面，利用 Restlet 来提供完整的 REST APIs，通过 m2eclipse 与 Eclipse 集成使用。支持 WebDAV 与 LDAP 安全身份认证。

私有仓库的使用如图 9-1 所示。

Public 为仓库组，包含了 central、3rd party、release 和 snapshots。Central 为代理仓库，通过代理访问中央仓库。3rd party 为宿主仓库，用于存放第三方的构件；release 和 snapshots 为宿主仓库，用于存放项目的构件。

开发人员进行日常开发时，通过 public 下载开发过程中所需的构件，绿色的 Maven 代表版本发布人员在版本发布时，直接将构件部署到项目仓库中。

3rd party 仓库的构件通过管理员在 Nexus 的控制台手工进行部署。

如果没有 Nexus 私服，我们所需的所有构件都需要通过 Maven 的中央仓库和第三方的 Maven 仓库下载到本地，而一个团队中的所有人都重复从 Maven 仓库下载构件无疑加大了仓库

的负载和浪费了外网带宽，如果网速慢，还会影响项目的进程。很多情况下项目的开发都是在内网进行的，连接不到 Maven 仓库怎么办呢？开发的公共构件怎么让其他项目使用？这个时候我们不得不为自己的团队搭建属于自己的 Maven 私服，这样既节省了网络带宽，又加速了项目搭建的进程，当然前提条件就是你的私服中拥有项目所需的所有构件。

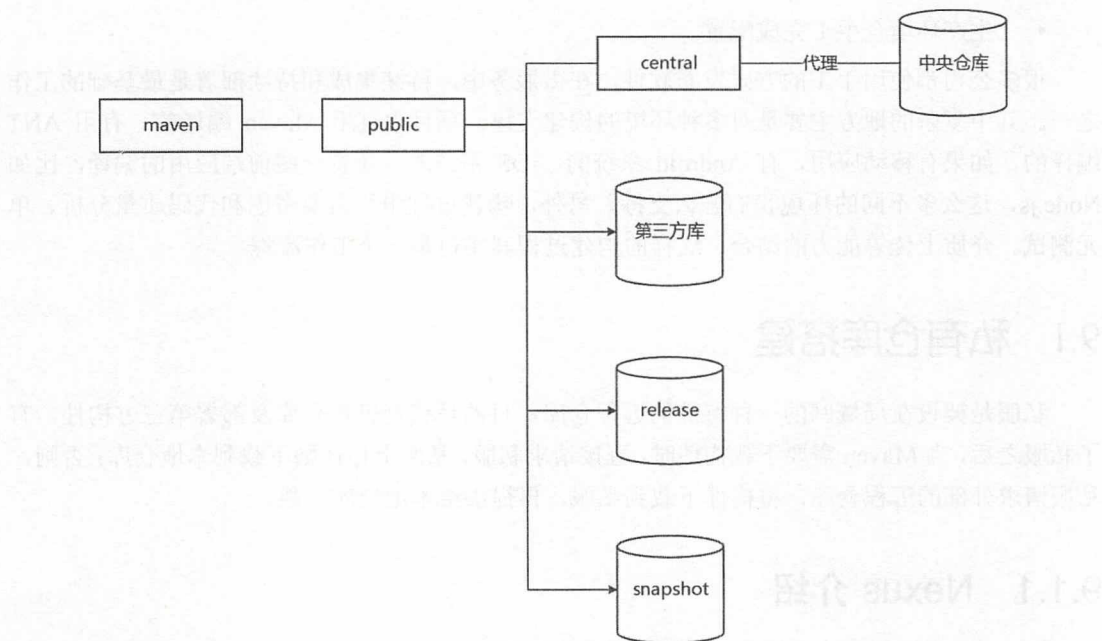


图 9-1 Nexus 私有仓库

9.1.2 安装与配置

从官网下载 Nexus，下载地址：<http://www.sonatype.org/nexus/go>。

Nexus 提供两种安装包，一种是包含 Jetty 容器的 bundle 包，另一种是不包含容器的 war 包。

如果下载的是 war 包，则必须自己另外配置 Servlet 容器（如 Tomcat）才能运行起来。

如果下载的是 bundle 包，则它自己内置了 Jetty，不用再下载其他依赖。

➤ 安装

解压安装包：

```
tar -zxvf nexus-3.0.0-03-unix.tar.gz
```


➤ 修改 JDK 路径

Nexus3.0.0 的版本需要 JDK1.8 的版本:

```
cd /data/nexus-3.0.0-03/bin
```

添加 JDK 配置:

```
INSTALL4J_JAVA_HOME_OVERRIDE=/data/jdk1.8.0_91
```

启动 Nexus 服务:

```
./ nexus
```

启动后打开浏览器输入: <http://XXX:8081>, 就能够跳转到 Nexus 的页面。

单击左侧菜单栏的 Repositories 按钮, 能够看到仓库中的信息。

- hosted 类型的仓库, 内部项目的发布仓库;
- Releases 是内部的模块中 release 模块的发布仓库;
- Snapshots 是发布内部的 SNAPSHOT 模块的仓库;
- 3rd party 是第三方依赖的仓库, 这个数据通常是由内部人员自行下载之后发布上去的;
- proxy 类型的仓库, 从远程中央仓库中寻找数据的仓库;
- group 类型的仓库, 用来方便开发人员进行设置的仓库。

9.1.3 在项目中使用

Nexus 搭建完成后, 我们可以把需要团队共享的 jar 包上传到私服中,

➤ 手工上传

在 Repository 列表中, 选择 3rd party→artifact upload→GAV Definition: GAV Parameters, 将 Auto Guess 打钩。在下方输入 JAR 包对应的 Group、Artifact、Version, Packaging 选择 JAR 格式。单击 “select Artifact(s) to upload” 按钮, 选择要上传的 JAR 包, 单击 “save” 保存。

➤ 工程部署

如果想实现在工程中直接上传到私服中, 在 “settings.xml” 的 “<mirrors>” 节点中增加:

```
<mirror>
  <id>ctxsdhy-hosted-release</id>
```

```

    <mirrorOf>*</mirrorOf>
    <name>ctxsdhy-hosted-release</name>
    <url>http://xxxx:8081/nexus/content/repositories/3rd-release/</url>
</mirror>
<mirror>
    <id>ctxsdhy-hosted-snapshot</id>
    <mirrorOf>*</mirrorOf>
    <name>ctxsdhy-hosted-snapshot</name>
    <url>http://xxxx:8081/nexus/content/repositories/3rd-release/</url>
</mirror>

```

在“<servers>”节点中增加登录的凭证，使用 Nexus 设置的用户名和密码：

```

<server>
    <id>ctxsdhy-hosted-snapshot</id>
    <username>admin</username>
    <password>123456</password>
</server>
<server>
    <id>ctxsdhy-hosted-release</id>
    <username>admin</username>
    <password>123456</password>
</server>

```

然后修改工程的 pom.xml，在其中加入如下代码：

```

<distributionManagement>
    <repository>
        <id>ctxsdhy-hosted-release</id>
        <name>ctxsdhy-hosted-release</name>
        <url> http://xxxx:8081/nexus/content/repositories/3rd-release/</url>
    </repository>
    <snapshotRepository>
        <id>ctxsdhy-hosted-snapshot</id>
        <name>ctxsdhy-hosted-snapshot</name>
        <url> http://xxxx:8081/nexus/content/repositories/3rd-release/</url>
    </snapshotRepository>
</distributionManagement>

```


id 要和 “settings.xml” 文件中的相对应。

然后编写代码后，在工程上单击右键，执行 Run as→Run configurations，选择工程，在 Goals 中输入 clern deploy，单击 Run。执行完成后，就可以在 Nexus 上看到上传的 JAR 文件。

最后在需要引用私有 JAR 包的工程中添加私服地址，加入上传的私有 JAR 包的相关配置就能够在项目中引用了。

9.2 Ansible

Ansible 是新出现的自动化运维工具，基于 Python 开发，集合了众多运维工具（puppet、cfengine、chef、func、fabric）的优点，实现了批量系统配置、批量程序部署、批量运行命令等功能。Ansible 是基于模块工作的，本身没有批量部署的能力。真正具有批量部署的是 Ansible 所运行的模块，Ansible 只是提供了一种框架。

- 连接插件 connection plugins：负责和被监控端实现通信；
- host inventory：指定操作的主机，是一个配置文件里面定义监控的主机；
- 各种模块核心模块、command 模块、自定义模块；
- 借助于插件完成记录日志邮件等功能；
- 通过 playbook 定制强大的配置、状态管理。

Ansible 的架构如图 9-2 所示。

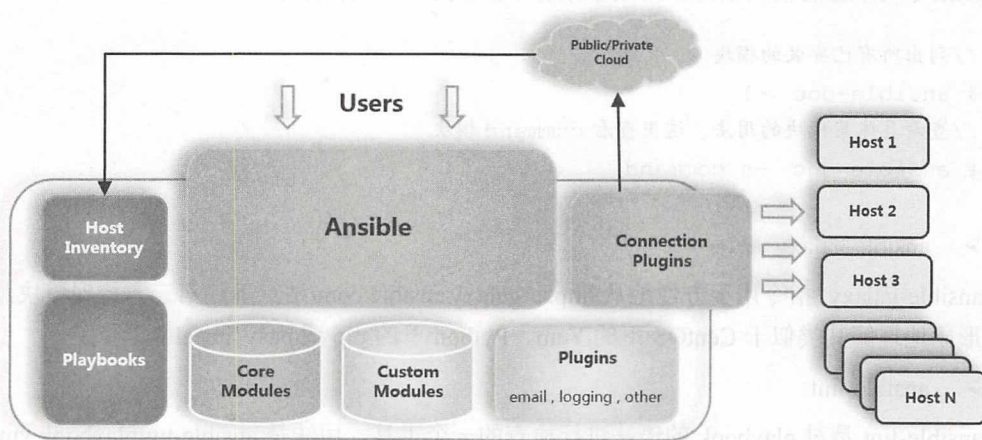


图 9-2 Ansible 架构

- Ansible：Ansible 的核心程序。

- **Host Inventory:** 记录了每一个由 Ansible 管理的主机信息，信息包括 SSH 端口、root 账号密码、IP 地址，等等。可以通过 file 来加载，也可以通过 CMDB 加载。
- **Playbooks:** YAML 格式文件，多个任务定义在一个文件中，使用时可以统一调用，“剧本”用来定义那些主机需要调用哪些模块来完成的功能。
- **Core Modules:** Ansible 执行任何管理任务都不是由 Ansible 自己完成，而是由核心模块完成的；Ansible 管理主机之前，先调用 core Modules 中的模块，然后指明管理 Host Inventory 中的主机就可以完成管理主机的操作。
- **Custom Modules:** 自定义模块，完成 Ansible 核心模块无法完成的功能，此模块支持任何语言编写。
- **Connection Plugins:** 连接插件，Ansible 和 Host 通信时使用。

Ansible 的七个指令

安装完 Ansible 后，发现 Ansible 一共提供了七个指令：ansible、ansible-doc、ansible-galaxy、ansible-lint、ansible-playbook、ansible-pull 和 ansible-vault。这里我们只查看 usage 部分，详细部分可以通过“指令 -h”的方式获取。

➤ ansible

ansible 是指令核心部分，其主要用于执行 ad-hoc 命令，即单条命令。默认后面需要跟主机和选项部分，默认不指定模块时，使用的是 command 模块。

➤ ansible-doc

该指令用于查看模块信息，常用参数有-l 和-s 两个，具体如下：

```
//列出所有已安装的模块
# ansible-doc -l
//查看具体某模块的用法，这里查看 command 模块
# ansible-doc -s command
```

➤ ansible-galaxy

ansible-galaxy 指令用于方便地从 <https://galaxy.ansible.com/> 站点下载第三方扩展模块，我们可以形象地理解其类似于 CentOS 下的 Yum、Python 下的 pip 或 easy_install。

➤ ansible-lint

ansible-lint 是对 playbook 的语法进行检查的一个工具。用法是 ansible-lintplaybook.yml。

➤ ansible-playbook

该指令是使用最多的指令，通过读取 playbook 文件后，执行相应的动作，后面会作为重点

来讲。

➤ ansible-pull

使用该指令需要谈到 Ansible 的另一种模式——pull 模式，这和我们平常经常用的 push 模式刚好相反，其适用于以下场景：有数量巨大的机器需要配置，即使使用非常高的线程还是要花费很多时间；要在一个没有网络连接的机器上运行 Ansible，比如在启动之后安装。这部分也会单独讲解。

➤ ansible-vault

ansible-vault 主要应用于配置文件中含有敏感信息，又不希望他能被人看到的场景，vault 可以加密/解密配置文件，属于高级用法。playbooks 里涉及配置密码或其他变量时，可以通过该指令加密，这样我们通过 cat 看到的会是一个密码串类的文件，编辑时需要输入事先设定的密码才能打开。这种 playbook 文件在执行时，需要加上--ask-vault-pass 参数，同样需要输入密码后才能正常执行。

9.3 持续集成

持续集成（Continuous integration，简称 CI）是一种软件开发实践，即团队开发成员经常集成它们的工作，通常每个成员每天至少集成一次，也就意味着每天可能会发生多次集成。每次集成都通过自动化的构建（包括编译、发布、自动化测试）来验证，从而尽快地发现集成错误。许多团队发现这个过程可以大大减少集成的问题，让团队能够更快地开发内聚的软件。

持续集成的目的与价值

持续集成的目的不是减少 build 失败的次数，而是尽早发现问题，在最短的时间内解决问题，减少风险和浪费。从而让产品开发流程更加敏捷，缩短产品开发周期，在产品上线后，让用户用得更加顺畅。

在没有应用持续集成之前，传统的开发模式是项目一开始就划分模块，每个开发人员分别负责一个模块，等所有的代码都开发完成之后再集成到一起提交给测试人员。随着软件技术队的发展，软件已经不能简单地通过划分模块的方式来开发，需要项目内部相互协作，划分模块这种传统的模式的弊端也越来越明显。由于很多 Bug 在项目早期的设计、编码阶段就引入，到最后集成测试时才发现问题，开发人员需要花费大量的时间来定位 Bug，加上软件的复杂性，Bug 的定位就更难了，甚至出现不得不调整底层架构的情况。这种情况的发生不仅仅对测试进度造成影响，而且会拖长整个项目周期。

而持续集成可以有效解决软件开发过程中的许多问题，在集成测试阶段之前就帮助开发人员发现问题，从而可以有效地确保软件质量，减小项目的风险，使软件开发团队从容地面对各

种变化。持续集成报告中可以体现目前项目进度，哪部分需求已经实现，哪些代码已经通过自动化测试，代码质量如何，让开发团队和项目组可以了解项目的真实状况。

持续集成的优点：

- 快速发现错误。每完成一点更新，就集成到主干，可以快速发现错误，定位错误也比较容易。
- 防止分支大幅偏离主干。如果不是经常集成，主干又在不断更新，则会导致以后集成的难度变大，甚至难以集成。

持续集成的一些原则：

- 所有的开发人员需要在本地机器上做本地构建，然后提交到版本控制库中，从而确保他们的变更不会导致持续集成失败。
- 开发人员每天至少向版本控制库中提交一次代码。
- 开发人员每天至少需要从版本控制库中更新一次代码到本地机器。
- 需要有专门的集成服务器来执行集成构建，每天要执行多次构建。
- 每次构建都要 100%通过。
- 每次构建都可以生成可发布的产品。
- 修复失败的构建是优先级最高的事情。

9.3.1 持续集成流程

持续集成工具如下。

- Apache Continuum: <http://continuum.apache.org/>。
- CruiseControl: <http://cruisecontrol.sourceforge.net/>。
- Hudson: <http://hudson-ci.org/>。
- Jenkins: <http://jenkins-ci.org/>。
- Luntbuild: <http://jenkins-ci.org/>。

经过比较，我们使用的是 Jenkins。

一般的持续集成流程如图 9-3 所示。

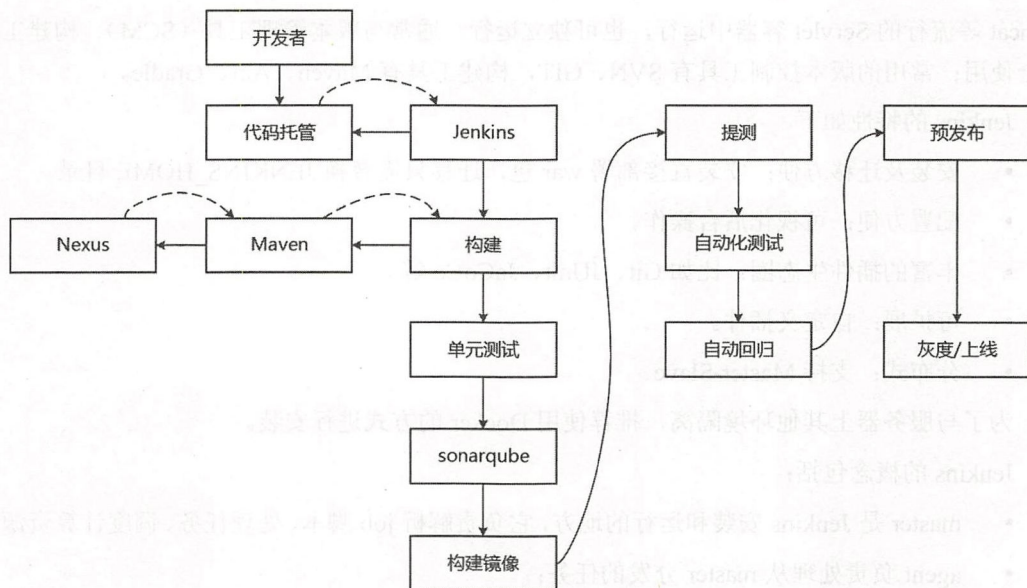


图 9-3 持续集成流程

(1) 开发人员完成需求的功能后，提交代码到代码仓库，Jenkins 从代码仓库（SVN 或者 Git）拉取项目分支代码。

(2) 执行构建，一般使用 Maven 进行项目的构建，当然也可以采用其他的构建工具。Maven 从 Nexus 私服摘取需要的 jar 包信息，完成编译。

(3) 编译完成后会自动触发单元测试，测试所有应用的单元测试代码。

(4) 单元测试通过后，触发 sonarcube 客户端执行静态代码检查，并且生成检查报告。

(5) 编译和构建完成后，需要生成 Docker 镜像。

(6) 将镜像发送到测试环境进行发布，触发自动化测试流程。

(7) 如果测试不通过，则触发回归流程，测试通过，将已经打包好的镜像推送到预发布环境。

(8) 预发布环境测试没有问题后，就可以进行灰度发布，将系统发布到线上环境。

在持续构建的过程中，能实时查看构建进度、构建状态、构建结果等详细信息。

9.3.2 Jenkins 介绍与安装

Jenkins 是一个开源的、提供友好操作界面的持续集成（CI）工具，起源于 Hudson，主要用于持续、自动的构建/测试软件项目、监控外部任务的运行。Jenkins 用 Java 语言编写，可在

Tomcat 等流行的 Servlet 容器中运行，也可独立运行。通常与版本管理工具（SCM）、构建工具结合使用；常用的版本控制工具有 SVN、GIT，构建工具有 Maven、Ant、Gradle。

Jenkins 的特性如下。

- 安装及迁移方便：安装直接部署 war 包，迁移只需替换 JENKINS_HOME 目录。
- 配置方便：可视化后台操作。
- 丰富的插件生态圈：比如 Git、JUnit、JaCoCo 等。
- 可扩展：自定义插件。
- 分布式：支持 Master-Slave。

为了与服务器上其他环境隔离，推荐使用 Docker 的方式进行安装。

Jenkins 的概念包括：

- master 是 Jenkins 安装和运行的地方，它负责解析 job 脚本、处理任务、调度计算资源；
- agent 负责处理从 master 分发的任务；
- executor 就是执行任务的计算资源，它可以在 master 或者 agent 上运行，多个 executor 也可以合作执行一些任务；
- job 任务用来定义具体的构建过程；
- Groovy 是一种基于 JVM（Java 虚拟机）的敏捷开发语言，它结合了 Python、Ruby 和 Smalltalk 的许多强大的特性，Groovy 代码能够与 Java 代码很好地结合，也能用于扩展现有代码，由于其运行在 JVM 上的特性，Groovy 可以使用其他 Java 语言编写的库，Jenkins 用 Groovy 作为 DSL；
- pipeline，流水线即代码（Pipeline as Code），通过编码而非配置持续集成/持续交付（CI/CD）运行工具的方式定义部署，流水线使得部署是可重现、可重复的；
- 流水线包括节点（node）、阶段（stage）和步骤（step）；
- 流水线执行在节点上。节点是 Jenkins 安装的一部分。流水线通常包含多个阶段，一个阶段包含多个步骤；
- node 在 Pipeline 的 context 中，node 是 job 运行的地方。node 会给 job 创建一个工作空间。工作空间就是一个文件目录，这是为了避免跟资源相关的处理互相产生影响。工作空间是 node 创建的，在 node 里的所有 step 都执行完毕后会自动删除；
- stage 阶段，stage 是一个任务执行过程的独立的并且唯一的逻辑块，pipeline 定义在语法上就是由一系列的 stage 组成的，每一个 stage 逻辑都包含一个或多个 step；
- step 步骤，一个 step 是整个流程中的一系列事情中的一个独立的任务，step 用来告诉

Jenkins 如何做：

- Jenkinsfile, Jenkins 支持创建流水线。它使用一种基于 Groovy 的流水线领域特定语言（Pipeline DSL）的简单脚本。而这些脚本，通常名字叫 Jenkinsfile。它定义了一些根据指定参数执行简单或复杂的任务的步骤。流水线创建好后，可以用来构建代码，或者编排从代码提交到交付过程中所需的工作。Jenkins 中的 Jenkinsfile 有点类似 Docker 中的 Dockfile。

构建的代码如下：

```
docker pull jenkinsci/Jenkins
```

镜像下载完成后，使用 docker run 运行镜像。

运行成功后，经过简单的设置，可以得到如图 9-4 所示的界面。

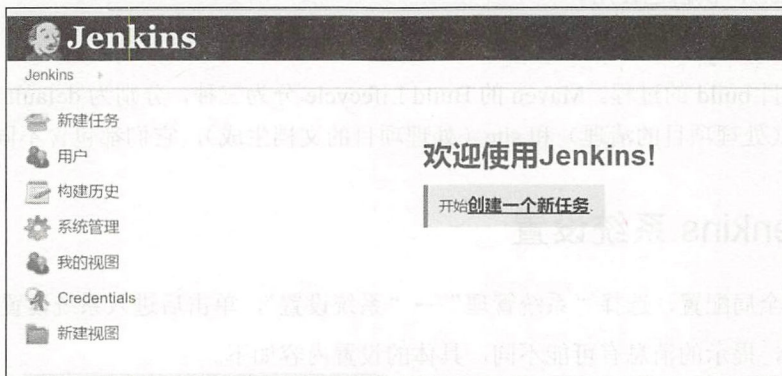


图 9-4 Jenkins 运行成功界面

9.3.3 Maven 介绍

Maven 是一个项目构建和管理的工具，提供了帮助管理构建、文档、报告、依赖、SCMS、发布、分发的方法。可以方便地编译代码、进行依赖管理、管理二进制库等。

Maven 的好处在于可以将项目过程规范化、自动化、高效化，以及具备强大的可扩展性。利用 Maven 自身及其插件还可以获得代码检查报告、单元测试覆盖率、实现持续集成等。Maven 的核心概念如下。

➤ pom

pom 是指 project object Model，pom 是一个 XML，在 Maven 2 里为 pom.xml，是 Maven 工

作的基础，在执行 task 或者 goal 时，Maven 会去项目根目录下读取 pom.xml 以获得需要的配置信息。

pom 文件中包含了项目的信息和 maven build 项目所需的配置。

➤ artifact

一个项目将要产生的文件，可以是 jar 文件、源文件、二进制文件、war 文件，甚至是 pom 文件。每个 artifact 都由 groupId:artifactId:version 组成的标识符唯一识别。需要被使用（依赖）的 artifact 都要放在仓库（见 Repository）中。

➤ repositories

repositories 是用来存储 Artifact 的。如果说我们的项目产生的 artifact 是一个个小工具，那么 Repositories 就是一个仓库，里面有我们自己创建的工具，也可以储存别人造的工具，我们在项目中需要使用某种工具时，在 pom 中声明 dependency，编译代码时就会根据 dependency 去下载工具（artifact），供自己使用。

➤ Build Lifecycle

指一个项目 build 的过程。Maven 的 Build Lifecycle 分为三种，分别为 default（处理项目的部署）、clean（处理项目的清理）和 site（处理项目的文档生成），它们都包含不同的 lifecycle。

9.3.4 Jenkins 系统设置

首先进行全局配置，选择“系统管理”→“系统设置”，单击后进入系统设置界面。

版本不同，提示的消息有可能不同，具体的设置内容如下。

➤ Jenkins Location 配置

Jenkins URL 项保持默认即可，填写系统管理员邮件地址（注意：如果不填写是发送不了邮件的，测试邮件发送时会报“553 Mail from must equal authorized user”错误）。

➤ 邮件通知配置

填写“SMTP 服务器”、“用户默认邮件后缀”，然后单击“高级”，勾选“使用 SMTP 认证”，填写邮箱用户名、密码和 SMTP 端口，接着勾选“通过发送测试邮件测试配置”，填写接收邮件的邮箱，单击“测试”，出现“Email was successfully sent”表明邮件通知配置成功。

➤ SSH remote hosts 配置

SSH 远程主机配置主要用来通过 SSH 方式远程发布。

➤ SonarQube servers 配置

配置质量检查服务。

➤ Maven 配置

配置 Maven 主要是配置 Maven 的 settings.xml 文件, Jenkins 在构建 Maven 项目时需要依靠该配置文件来执行 Maven, 其配置如下:

➤ JDK 配置

选择本机配置的 JDK 版本。

➤ SonarQube Scanner 配置

质量检查脚本设置。

9.3.5 集成 Sonar

通过 Maven 进行集成

修改 Maven 的主配置文件 (\${MAVEN_HOME}/conf/settings.xml 文件或者 ~/.m2/settings.xml 文件), 在其中增加访问 Sonar 数据库及 Sonar 服务地址, 添加如下配置:

```
<profile>
<id>sonar</id>
<properties>
    <sonar.jdbc.url>jdbc:mysql://localhost:3306/sonar</sonar.jdbc.url>
    <sonar.jdbc.driver>com.mysql.jdbc.Driver</sonar.jdbc.driver>
    <sonar.jdbc.username>sonar</sonar.jdbc.username>
    <sonar.jdbc.password>sonar</sonar.jdbc.password>
    <sonar.host.url>http://localhost:9000</sonar.host.url> <!-- Sonar 服务
器访问地址 -->
</properties>
</profile>
<activeProfiles>
    <activeProfile>sonar</activeProfile>
</activeProfiles>
```

注意, 此处 sonar.host.url 地址应根据 Sonar 的部署情况修改。

同样, 为了避免内存溢出, 推荐增加内存堆栈的大小。设置 MAVEN_OPTS 环境变量:

```
set MAVEN_OPTS="-Xmx512m -XX:MaxPermSize=256m"
```

使用 Sonar:

- (1) 运行 Sonar 服务器。
- (2) 通过 mvn sonar: sonar 将代码注入 Sonar 中进行分析处理，并将处理结果以 XML 的形式保存在数据库中。
- (3) 通过浏览器访问，显示分析结果。
- (4) 持续运行 Maven 构建，会迭代显示分析结果。
- (5) 可以显式指定 Sonar 插件的版本。

```
<project>
  <build>
    <plugins>
      <plugin>
        <groupId>org.codehaus.sonar</groupId>
        <artifactId>sonar-maven-plugin</artifactId>
        <version>3.5.1</version>
      </plugin>
    </plugins>
  </build>
</project>
```

可以显式地将 Sonar 绑定到 Maven 生命周期中：

```
<plugin>
  <groupId>org.codehaus.sonar</groupId>
  <artifactId>sonar-maven-plugin</artifactId>
  <version>3.5.1</version>
  <executions>
    <execution>
      <id>sonar</id>
      <phase>site</phase>
      <goals>
        <goal>sonar</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

此时，指定 Maven 的 site 声明周期时，会自动调用 sonar.sonar 命令。

直接与 Jenkins 集成

在 Jenkins 的插件管理中选择安装 sonar jenkins plugin, 该插件可以使项目每次构建都调用 Sonar 进行代码度量。

进入配置页面对 Sonar 插件进行配置。

配置构建项目, 增加 Post Build Action: 应用程序在构建时就会自动触发 Sonar 对代码的检查。

9.3.6 构建工程

新建一个 Maven 项目, 需要为新的构建任务指定一个名称, 如图 9-5 所示。



图 9-5 Jenkins 构建工程页面

然后进行项目配置, 指定源代码的位置。可以使用 SVN, 也可以使用 Git。

构建触发器

Jenkins 提供了 6 种构建触发器, 分别是:

- build whenever a snapshot dependency is built, 当 job 依赖的快照版本被 “build” 时, 执行本 job;
- 触发远程构建 (例如, 使用脚本);
- build after other projects are built, 当本 job 依赖的 job 被 “build” 时, 执行本 job;
- build periodically, 隔一段时间 “build” 一次, 不管版本库代码是否发生变化, 通常不会采用此种方式;
- GitHub hook trigger for GITScm polling, 通过 GitHub 钩子触发;
- poll scm, 隔一段时间比较一次源代码, 如果发生变更, 那么就进行 build。否则, 不进行 “build”, 通常采用这种方式。

构建 (Build)

调用 Maven 的 “clean install -Dmaven.test.skip=true” 命令。

Poll SCM: 这是 CI 系统中常见的选项。选择此选项时, 可以指定一个定时作业表达式来定义 Jenkins 每隔多久检查一次源代码仓库的变化。如果发现变化, 就执行一次构建。例如, 表达式中填写 0,15,30,45 * * * * 将使 Jenkins 每隔 15 分钟就检查一次源码仓库的变化。

Build periodically: 此选项仅仅通知 Jenkins 按指定的频率对项目进行构建, 而不管 SCM 是否有变化。

选择 Add post-build action, 然后选择 Deploy war/ear to a container。

需要配置 WAR/EAR files: war 文件的存放位置, 如 target/test.war (注意: 相对路径, target 前是没有 “/” 的)。

- Context path: 访问时需要输入的内容, 例如, ofCard 访问时 “http://192.168.x.x:8080/ofCard/” 如果为空, 则默认是 war 包的名字。
- Container: 选择你的 Web 容器, 如 Tomcat 8.x。
- Manager user name: 填入 tomcat-users.xml 配置的 username 内容。
- Manager password: 填入 tomcat-users.xml 配置的 password 内容。
- Tomcat URL: 填入 http://192.168.x.x:8080/。
- Deploy on failure: 构建失败依然部署, 一般不选择。

设置完成后, 进行保存操作。

当任务一旦运行, 会看到这个任务正在队列中的仪表板和当前工作主页上运行。

当前作业主页上还包含了一些有趣的条目。左侧栏的链接主要控制 Job 的配置、删除作业、构建作业, 右边部分的链接指向最新的项目报告和构件。

单击构建历史（Build History）中某个具体的构建链接，就能跳转到 Jenkins 为这个构建实例创建的构建主页上。

按照以上步骤，可以成功完成自动化部署环境的搭建。

9.3.7 配置测试

因为要执行测试，所以执行以下步骤：

- 创建一个自由风格的 Jenkins 项目。
- 在构建一栏，输入以下脚本，用于触发 newman 测试。

```
newman run /home/zhangfeng/foobar.json --reporters cli,html,json,junit
--reporter-html-export /home/zhangfeng/foobar_report.html
```

- 在主项目的构建配置中，增加构建后操作，构建完成后自动触发执行这个测试项目。

脚本执行完成后，可以在 Jenkins 的日志中查看结果，也可以以网页的方式查看结果。

9.4 灰度发布

产品的发布大多是功能完成就直接上线，替换了原有的版本，这种跳跃式的发布是非常危险的，如果产品影响面大，则对项目成员的压力是非常大的。灰度发布是在发布新版本时，先切分部分流量给新版本，稳定了之后再切分所有流量到新版本。这样一旦出现问题，马上修改切分的流量就可以，不需要重新发布，减少了发布风险。这种基于 A/B 测试分流的灰度发布方式已经成为很多公司发布的一个必经流程。

灰度就是不希望所有人都看到，就是为了控制影响范围，之所以要做这种限制，就说明发布的人心里对这个发布的版本的稳定性是不确定的，害怕影响范围太大风险不可控。也就是说，这个风险因素在开发和测试环境都没有办法控制，只能在生产环境中观察。

灰度发布常见一般有三种方式。

➤ 通过调节负载均衡权重

负载均衡建立在现有网络结构之上，它提供了一种廉价、有效、透明的方法来扩展网络设备和服务器的带宽，增加吞吐量，加强网络数据处理能力，提高网络的灵活性和可用性。

负载均衡，英文名称为 Load Balance，其意思就是分摊到多个操作单元上执行。例如，Web 服务器、FTP 服务器、企业关键应用服务器和其他关键任务服务器等，从而共同完成工作任务。

简单配置代码如下：

```
http {
    upstream cluster {
        ip_hash; #如果你的系统中没有使用第三方缓存管理工具，建议使用此方式
        server 192.168.1.210:80 weight=5;
        server 192.168.1.211:80 weight=3;
        server 192.168.1.212:80 weight=1;
    }

    server {
        listen 80;

        location / {

            proxy_next_upstream    error timeout;
            proxy_redirect          off;
            proxy_set_header       Host $host;
            #proxy_set_header       X-Real-IP $remote_addr;
            proxy_set_header       X-Real-IP $http_x_forwarded_for;
            proxy_set_header       X-Forwarded-For $proxy_add_x_forwarded_for;
            client_max_body_size    100m;
            client_body_buffer_size 256k;
            proxy_connect_timeout  180;
            proxy_send_timeout     180;
            proxy_read_timeout     180;
            proxy_buffer_size       8k;
            proxy_buffers           8 64k;
            proxy_busy_buffers_size 128k;
            proxy_temp_file_write_size 128k;
            proxy_pass http://cluster;
        }
    }
}
```

这种方式灰度发布通过 weight 实现的，但这种方式只适合修改节点的行为，而且要求应用都是一模一样的。其实质作用是，节点增加或删除之后，对负载能力进行调节，目的是为了让流量最终保持均衡。

➤ 使用 HTTP 头信息判断+权重

HTTP 请求传输过程中，会自动带上 User-Agent、Host、Referer、Cookie 等信息。我们只需要判断 IP 地址段、用户代理、Cookie 中的信息即可。

➤ 使用 Nginx+Lua 实现 Web 项目的灰度发布

使用 Nginx+Lua 模块，这种方式适合很多场景，非常强大，前提是需要熟悉 Lua 的语法。示例代码如下：

```
location / {
    content_by_lua '
        myIP = ngx.req.get_headers()["X-Real-IP"]
        if myIP == nil then
            myIP = ngx.req.get_headers()["x_forwarded_for"]
        end
        if myIP == nil then
            myIP = ngx.var.remote_addr
        end
        if myIP == "出口 IP" then
            ngx.exec("@client")
        else
            ngx.exec("@client_test")
        end
    ';
}
```

```
location @client{
    proxy_next_upstream    error timeout;
    proxy_redirect         off;
    proxy_set_header       Host $host;
    #proxy_set_header       X-Real-IP $remote_addr;
    proxy_set_header       X-Real-IP $http_x_forwarded_for;
    proxy_set_header       X-Forwarded-For $proxy_add_x_forwarded_for;
    client_max_body_size    100m;
    client_body_buffer_size 256k;
    proxy_connect_timeout   180;
    proxy_send_timeout      180;
    proxy_read_timeout      180;
    proxy_buffer_size       8k;
```

```
proxy_buffers      8 64k;
proxy_busy_buffers_size 128k;
proxy_temp_file_write_size 128k;
proxy_pass http://client;
}

location @client_test{
    proxy_next_upstream    error timeout;
    proxy_redirect         off;
    proxy_set_header       Host $host;
    #proxy_set_header      X-Real-IP $remote_addr;
    proxy_set_header       X-Real-IP $http_x_forwarded_for;
    proxy_set_header       X-Forwarded-For $proxy_add_x_forwarded_for;
    client_max_body_size   100m;
    client_body_buffer_size 256k;
    proxy_connect_timeout  180;
    proxy_send_timeout     180;
    proxy_read_timeout     180;
    proxy_buffer_size       8k;
    proxy_buffers           8 64k;
    proxy_busy_buffers_size 128k;
    proxy_temp_file_write_size 128k;
    proxy_pass http://client_test;
}
```

9.5 小结

Nexus 能够实现私有仓库的搭建，工程中经常使用的包或者公司内部的包都可以使用此软件进行维护，能够有效地提高团队的协作效率。

Ansible 是一种自动化运维工具，当机器数量达到一定的规模后，如果是每台机器都单独维护，则是一件成本非常高的事情，借助 Ansible 可以有效地提高运维效率，降低运维成本。

持续集成和部署是微服务中非常重要的一个环节，因为如果没有自动化发布，则会给整个团队带来非常大的负担，所以这件事情一般建议在团队构建的初期完成，保证所有项目都能在自动化发布平台上运行，而不是所有的项目在出现问题时才与运维进行沟通交互。

10 chapter

第 10 章

微服务之日志收集与监控

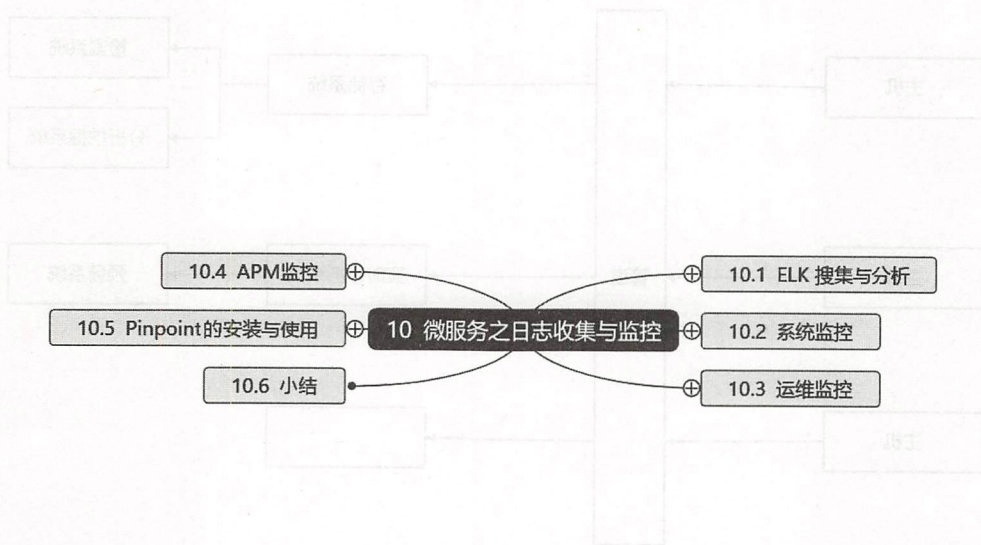


图 10-1 微服务之日志收集与监控

微服务的特点决定了功能模块的部署是分布式的，以往在单应用环境下，所有的业务都在同一个服务器上，如果服务器出现错误和异常，我们只要盯住一个点，就可以快速定位和处理问题。但在微服务的架构下，大部分功能模块都是单独部署运行的，彼此通过总线交互，都是无状态的服务，这种架构下，前后台的业务流会经过很多个微服务的处理和传递，我们难免会遇到这样的问题：

- 分散在各个服务器上的日志怎么处理？
- 如果业务流出现了错误和异常，如何定位是哪个点出的问题？
- 如何快速定位问题？
- 如何跟踪业务流的处理顺序和结果？

以前在单应用下的日志监控很简单，在微服务架构下却成了一个大问题，如果无法跟踪业务流，无法定位问题，则我们将耗费大量的时间来查找和定位问题，在复杂的微服务交互关系中，我们就会非常被动。微服务日志监控流程如图 10-1 所示。

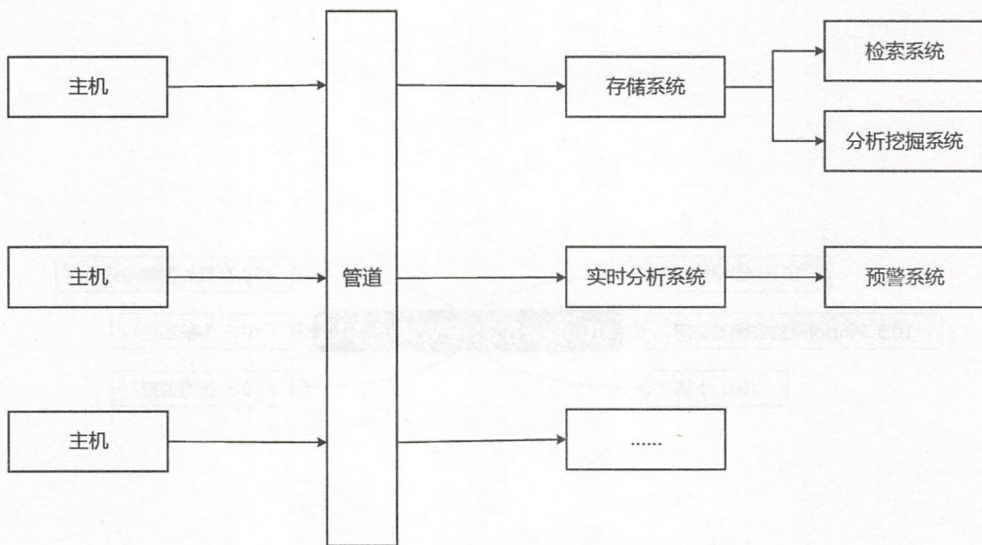


图 10-1 微服务日志监控流程

从左至右，先是从宿主服务器上去收集日志，紧接着是一个管道，负责数据传输，接着数据被分流，一部分进了存储系统，比如 Elasticsearch、Hadoop，一部分进了实时分析系统，比如 Storm、Spark Streaming。这个架构已经被很多互联网公司采用。但这个架构设计有几个地方值得大家关注：

- 管道除了传输，还有很多能力，最常用的是简单预处理和数据缓冲。
- 传输可以是多级，一般来说规模大到一定程度时，数据往一个点的海量汇聚很容易出

问题，分级处理是个很好的方式。

- 日志收集方式尽量统一，很多设计里对于业务日志、容器日志（如 Docker）、宿主机日志等采用不同 agent，这是一个不好的设计方式，agent 也要管理，引入太多 agent 只会给自己增加不必要的麻烦。

微服务监控的挑战：

监控的目的是为了让集群中所有的服务组件，不管是 HTTP 服务、数据库服务，还是中间件服务，都能够健康稳定地运行，能发现问题，遇到问题能找到原因。

过去，监控工具侧重于基础设施或单一软件组件，用于衡量运营健康。这些工具在实现这一目标方面只取得了一定的成功，但是对于单一的、传统的应用程序和基础设施来说效果不错。微服务的出现暴露了工具中的弱点。

现在，组件托管在位于私有云、公共云或两者的混合体之间的虚拟化机器或容器内。并不需要关心服务 CPU 用了多少，内存用了多少。只需要确保这些服务相互通信以提供所需的结果即可。需要从监控的角度看几件事情：

- 微服务集群中是否所有的服务的吞吐率、响应时间都正常？
- 服务调用线中哪些线负载过大，哪些线负载过小？
- 服务的错误率，如 HTTP 500 错误。

10.1 ELK 搜集与分析

ELK 是 Elasticsearch、Logstash 和 Kibana 的简称，这三套开源工具组合起来能搭建一套强大的集中式日志管理平台。

Elasticsearch 是一个开源的分布式搜索引擎，提供搜索、分析、存储数据三大功能。它的特点有：分布式、自动发现、索引自动分片、索引副本机制、RESTful 风格接口、多数据源及自动搜索负载等。

Logstash 是一个开源的用来收集、解析、过滤日志的工具，支持几乎任何类型的日志，包括系统日志、业务日志和安全日志。它可以从许多来源接收日志，这些来源主要包括 Syslog、消息传递（例如，RabbitMQ）和 Filebeat；能够以多种方式输出数据，这些方式主要包括电子邮件、WebSockets 和 Elasticsearch。

Kibana 是一个基于 Web 的友好图形界面，用于搜索、分析和可视化存储在 Elasticsearch 中的数据。它利用 Elasticsearch 的 RESTful 接口来检索数据，不仅允许用户定制仪表板视图，还允许他们以特殊的方式查询、汇总和过滤数据。

10.1.1 工作流程

在需要收集日志的所有服务上部署 logstash，作为 logstash agent (logstash shipper) 用于监控并过滤收集日志，将过滤后的内容发送到 Redis，然后 logstash indexer 将日志收集在一起交给全文搜索服务 Elasticsearch，可以用 Elasticsearch 进行自定义搜索，通过 Kibana 结合自定义搜索进行页面展示。Logstash 收集 AppServer 产生的 Log，并存放放到 Elasticsearch 集群中，而 Kibana 则从 ES 集群中查询数据生成图表，再返回给 Browser。

10.1.2 日志格式

日志收集主要分为访问日志和业务日志两部分。

➤ 访问日志

访问日志用来记录用户的访问轨迹，主要有两种收集方式，一种是通过服务器端的日志，比如 Nginx 日志，还有一种是使用客户端埋点技术，对需要收集的数据进行埋点，然后进行统一收集。

Nginx 的日志格式可以通过配置进行设置，示例如下：

```
api.yejianfeng.com 10.171.xx.xx 100.97.xx.xx [10/Jun/2015:10:53:24 +0800]
"POST /api1.2/user/mechanicupdate/ HTTP/1.0" start_time=1276099200&lng=
110.985723&source=android&uid=328910&lat=35.039471&city=140800 200 754 "-" "-"
0.161 0.159
```

客户端埋点使用的最多的就是百度统计或者是 Google 统计，也可以根据自身需求进行数据的格式化，然后统一收集、分析。百度统计的示例代码如下：

```
<script>
var _hmt = _hmt || [];
(function() {
  var hm = document.createElement("script");
  hm.src = "https://hm.baidu.com/hm.js?c18c3033b4f71f19f4ace2f1d2baff88";
  var s = document.getElementsByTagName("script")[0];
  s.parentNode.insertBefore(hm, s);
})();
</script>
```

将代码放到要收集的页面上，服务端对数据进行采集。

➤ 业务日志

业务日志一般是业务应用的日志，可以是多种语言的，如果是 Java，通常使用 log4j2 进行日志的打印。

对于日志的收集，一般要求相同的业务使用同一种日志格式，便于收集整理。

Log4j2 的日志输出格式如下。

- %d{yyyy-MM-dd HH:mm:ss,SSS}: 日志生产时间，输出到毫秒的时间；
- %-5level: 输出日志级别，-5 表示左对齐并固定输出 5 个字符，如果不足则在右边补 0；
- %c: logger 的名称 (%logger)；
- %t: 输出当前线程名称；
- %p: 日志输出格式；
- %m: 日志内容，即 logger.info("message")；
- %n: 换行符；
- %C: Java 类名 (%F)；
- %L: 行号；
- %M: 方法名；
- %l: 输出语句所在的行数，包括类名、方法名、文件名、行数；
- hostName: 本地机器名；
- hostAddress: 本地 IP 地址。

10.1.3 平台搭建

安装主要有 6 个步骤，软件的安装地址为 <https://www.elastic.co/start>。

➤ 安装 Elasticsearch

首先到官网下载最新版本的 Elasticsearch 压缩包：

<https://artifacts.elastic.co/downloads/elasticsearch/elasticsearch-6.0.0.zip>。

下载后解压缩，进入 ./bin 目录下，执行 elasticsearch.bat。启动成功后，输入 <http://localhost:9200/?pretty>，得到如下信息则表示安装成功。

```
{
  "name" : "fbbNk-R",
  "cluster_name" : "elasticsearch",
```

```

"cluster_uuid" : "e7ek-shVSxG0c2G76x716g",
"version" : {
  "number" : "6.0.0",
  "build_hash" : "8f0685b",
  "build_date" : "2017-11-10T18:41:22.859Z",
  "build_snapshot" : false,
  "lucene_version" : "7.0.1",
  "minimum_wire_compatibility_version" : "5.6.0",
  "minimum_index_compatibility_version" : "5.0.0"
},
"tagline" : "You Know, for Search"
}

```

➤ 安装 Kibana

下载最新版本的 Kibana 压缩包。

下载地址: https://artifacts.elastic.co/downloads/kibana/kibana-6.0.0-windows-x86_64.zip。

解压缩, 然后进入 ./bin 目录, 启动 kibana.bat。

➤ 安装 X-Pack

X-Pack 是 Elasticsearch 的一个扩展包, 将安全、警告、监视、图形和报告功能捆绑在一个易于安装的软件包中。

进入 Elasticsearch 的 bin 目录下, 执行如下命令:

```
elasticsearch-plugin.bat install x-pack
```

然后设置密码, 执行命令如下:

```
x-pack/setup-passwords.bat auto
```

进入 Kibana 的 bin 目录, 执行如下命令:

```
kibana-plugin.bat install x-pack
```

安装完成后, 进入 Kibana 的 config 中, 修改 kibana.yml:

```

elasticsearch.username: "kibana"
elasticsearch.password: "<pwd>"

```


<pwd>是使用 setup-passwords 生成的密码。

重新启动 Elasticsearch 和 Kibana。

打开浏览器，输入 <http://localhost:5601/>，可以打开 Kibana，此时需要输入用户名和密码登录，进入 Kibana 界面，如图 10-2 所示。

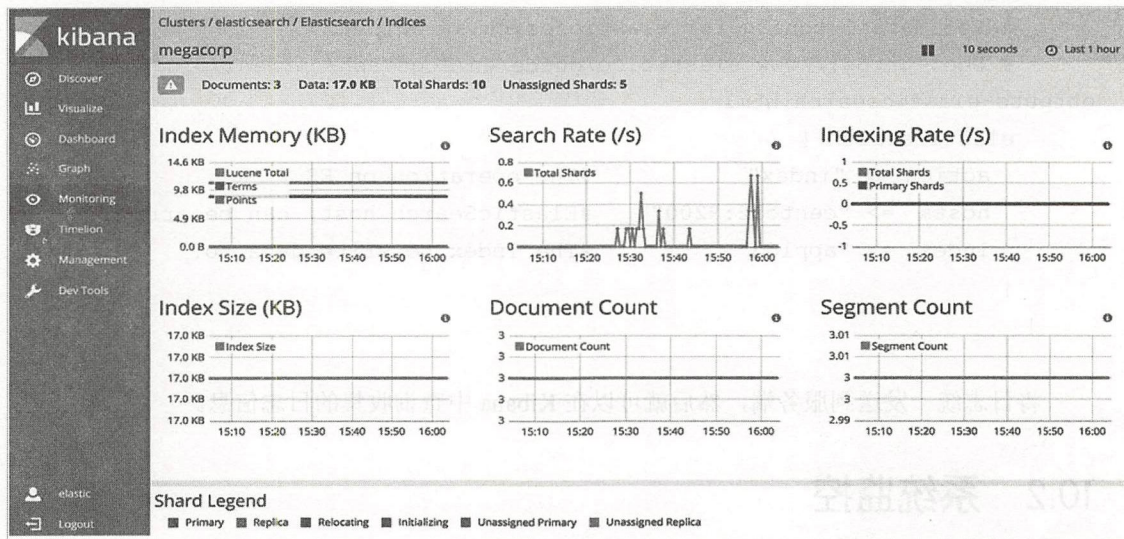


图 10-2 Kibana 界面

➤ 安装 Logstash agent

首先安装 Java 环境，然后下载 Logstash，下载地址如下：

<https://artifacts.elastic.co/downloads/logstash/logstash-6.0.1.zip>。

下载完成后解压缩。

在 Config 目录下编写配置文件 log4j_to_es.conf，代码如下：

```
input {
  # For detail config for log4j as input,
  # See: https://www.elastic.co/guide/en/logstash/current/plugins-inputs-
log4j.html
  log4j {
    mode => "server"
    host => "centos2"
    port => 4567
```

```

    }
  }
  filter {
    #Only matched data are send to output.
  }
  output {
    # For detail config for elasticsearch as output,
    # See: https://www.elastic.co/guide/en/logstash/current/plugins-
    outputs-elasticsearch.html
    elasticsearch {
      action => "index"           #The operation on ES
      hosts  => "centos2:9200"    #ElasticSearch host, can be array.
      index  => "applog"          #The index to write data to.
    }
  }
}

```

将日志统一发送到服务端，然后就可以在 Kibana 中查询收集的日志信息。

10.2 系统监控

系统监控是微服务中非常重要的一个环节，系统的健康与否直接关系到整体应用的使用情况。如果没有监控，就不能够及时掌握线上应用程序的应用情况，不能够及时发现问题，很多问题往往要到客户处才能够发现，使用户对产品失去信心，这有可能导致非常严重的后果。

10.2.1 监控策略和监控对象

对于监控来说，监控策略非常重要，没有好的策略标准，很可能导致“胡子眉毛一把抓”，重要的和不重要的问题一起处理，本来很紧急的事被延误。下面我们看一下通常用到的监控策略都有哪些。

➤ 定义告警优先级策略

一般的监控到的结果是成功或者失败，如 Ping 不通、访问网页出错、连接不到 Socket。这些称之为故障，故障是最优先的告警。

➤ 定义告警信息内容标准

当 server 或应用发生问题时告警信息内容有许多，如告警执行业务名称、server IP、监控的线路、监控的服务错误级别、出错信息、发生时间等。如果全部收集可能非常杂乱，所以需要

分门别类对其进行处理。

➤ 统一接收汇总报表

一般是使用统一的服务器对监控数据进行汇总，然后定时对汇总结果进行统一处理。

➤ 定义故障告警主次

对于监控同一台 server 的服务，需要定义一个主要监控对象。这样既能大大降低告警消息数量，又让监控更加合理、更加有效率。

➤ 实现对常见性故障业务的自我修复功能

实现对常见性故障业务自我修复功能脚本进行统一部署并对修复后故障进行检查，告警检查频次不多于 3 次。

➤ 设定监控范围和目标

实现对负载均衡设备、网络设备、server、存储设备、安全设备、数据库、中间件及应用软件等 IT 资源的全面监控管理；同一时间自己主动收集、过滤、关联和分析各种管理功能产生的故障事件。实现对故障的提前预警和高速定位。对网络和业务应用等 IT 资源的性能进行监控，定期提供性能报表和趋势报表，为性能优化及未来系统扩容提供科学根据。

那么监控的对象又包括哪些呢？

通常情况下，我们能够将监控对象这么来分：

- server 监控，主要监控 server 信息，如 CPU 负载、内存使用率、磁盘使用率、登录用户数、进程状态、网卡状态等。
- 应用程序监控，主要监控该应用程序的服务状态、吞吐量和响应时间，由于不同应用需要监控的对象不同，这里不一一列举了。
- 数据库监控，一般监控数据库状态，数据库表或者表空间的使用情况，是否有死锁，错误日志、性能信息，等等。
- 网络监控，主要监控当前的网络状况、网络流量等。

10.2.2 进程监控

进程的监控主要是对线上的应用程序进行监控，可能是类似 Nginx、Tomcat 等这种应用，也可能是自己开发的应用程序。

下面以一个监控 Nginx 的 Python 脚本来举例，脚本代码如下：

```
#!/usr/bin/env python
import os, sys, time
```

```
while True:
    time.sleep(3)
    try:
        ret = os.popen('ps -C nginx -o pid,cmd').readlines()
        if len(ret) < 2:
            print "nginx process killed, restarting service in 3 seconds."
            time.sleep(3)
            os.system("service nginx restart")
    except:
        print "Error", sys.exc_info()[1]
```

程序会每隔 3 秒检查一下 Nginx 进程的状态，如果发现进程出现问题，则会重新启动 Nginx 服务，保证 Nginx 进程的可用性。

其他进程的监控方式与此类似。

10.2.3 数据波动监控

数据波动需要对于公司的现有业务有一定的了解，也就是说，公司的数据化程度非常高。

对于网站来说，每个频道每天的 PV、UV、DAU 等相关的数据都应了解，如果出现暴增等异常情况，能够及时发送邮件进行告警。

对于数据监控的示例如图 10-3 所示。

	浏览量(PV) ↓	访客数(UV)	IP数	跳出率	平均访问时长
今日	21,791,443	1,494,093	1,127,324	41.64%	00:20:58
昨日	23,537,834	1,730,547	1,264,958	41.02%	00:22:56
预计今日	23,244,949 ↓	1,558,443 ↓	1,180,577 ↓	--	--

图 10-3 数据波动监控

10.2.4 常用监控命令

当系统出现问题，我们在定位问题时，往往需要一些常用的监控命令来进行辅助分析。

常用的监控命令如下。

➤ top

top 命令是 Linux 下常用的性能分析工具，能够实时显示系统中各个进程的资源占用状况，

类似于 Windows 的任务管理器。

`top` 显示系统当前的进程和其他状况，是一个动态显示过程，即可以通过用户按键来不断刷新当前状态。如果在前台执行该命令，它将独占前台，直到用户终止该程序为止。比较准确地说，`top` 命令提供了实时对系统处理器的状态监视，它将显示系统中 CPU 最“敏感”的任务列表。该命令可以按 CPU 使用、内存使用和执行时间对任务进行排序；而且该命令的很多特性都可以通过交互式命令或者在个人定制文件中进行设定。

➤ Top（选项）

选项

- b: 以批处理模式操作；
- c: 显示完整的命令；
- d: 屏幕刷新间隔时间；
- I: 忽略失效过程；
- s: 保密模式；
- S: 累积模式；
- i<时间>: 设置间隔时间；
- u<用户名>: 指定用户名；
- p<进程号>: 指定进程；
- n<次数>: 循环显示的次数。

top 交互命令

在 `top` 命令执行过程中可以使用的一些交互命令。这些命令都是单字母的，如果在命令行中使用了 `-s` 选项，其中一些命令可能会被屏蔽。

- h: 显示帮助画面，给出一些简短的命令总结说明；
- k: 终止一个进程；
- i: 忽略闲置和僵死进程，这是一个开关式命令；
- q: 退出程序；
- r: 重新安排一个进程的优先级；
- S: 切换到累计模式；

s: 改变两次刷新之间的延迟时间（单位为 s），如果有小数，则换算成 ms。输入 0 值则系统将不断刷新，默认值是 5s；

f 或者 F: 从当前显示中添加或者删除项目;

o 或者 O: 改变显示项目的顺序;

l: 切换显示平均负载和启动时间信息;

m: 切换显示内存信息;

t: 切换显示进程和 CPU 状态信息;

c: 切换显示命令名称和完整命令行;

M: 根据驻留内存大小进行排序;

P: 根据 CPU 使用百分比大小进行排序;

T: 根据时间/累计时间进行排序;

w: 将当前设置写入 ~/.toprc 文件中。

➤ iostat

iostat 主要用于监控系统设备的 I/O 负载情况, iostat 首次运行时显示自系统启动开始的各项统计信息, 之后运行 iostat 将显示自上次运行该命令以后的统计信息。用户可以通过指定统计的次数和时间来获得所需的统计信息。

使用方式: iostat -d -k 2

- -d, 显示设备(磁盘)使用状态;
- -k, 某些使用 block 为单位的列强制使用 Kilobytes 为单位;
- 2, 数据显示每隔 2 秒刷新一次。

```
iostat -d -k 1 10
```

Device:	tps	kB_read/s	kB_wrtn/s	kB_read	kB_wrtn
sda	39.29	21.14	1.44	441339807	29990031
sda1	0.00	0.00	0.00	1623	523
sda2	1.32	1.43	4.54	29834273	94827104
sda3	6.30	0.85	24.95	17816289	520725244
sda5	0.85	0.46	3.40	9543503	70970116
sda6	0.00	0.00	0.00	550	236
sda7	0.00	0.00	0.00	406	0
sda8	0.00	0.00	0.00	406	0
sda9	0.00	0.00	0.00	406	0
sda10	60.68	18.35	71.43	383002263	1490928140

➤ vmstat

vmstat 是最常见的 Linux/UNIX 监控工具，可以展现给定时间间隔的服务器的状态值，包括服务器的 CPU 使用率、内存使用、虚拟内存交换情况、I/O 读写情况。这个命令是笔者查看 Linux/UNIX 时最喜爱的命令，一个是 Linux/UNIX 都支持，二是相比 top，可以看到整个机器的 CPU、内存、I/O 的使用情况，而不是单单看到各个进程的 CPU 使用率和内存使用率。

实际上，在应用过程中，我们会在一段时间内一直监控：

```
vmstat 2
procs -----memory----- ---swap-- ----io---  -system--  ---cpu---
r  b   swpd   free   buff  cache   si   so   bi   bo   in   cs   us   sy   id   wa
1  0       0 3499840 315836 3819660 0    0    0    1    2    0    0    0   100    0
0  0       0 3499584 315836 3819660 0    0    0    0   88   158    0    0   100    0
0  0       0 3499708 315836 3819660 0    0    0    2   86   162    0    0   100    0
0  0       0 3499708 315836 3819660 0    0    0   10   81   151    0    0   100    0
1  0       0 3499732 315836 3819660 0    0    0    2   83   154    0    0   100    0
```

参数的含义：

- r，表示运行队列（就是说多少个进程真的分配到 CPU），笔者测试的服务器目前 CPU 比较空闲，没什么程序在跑，当这个值超过了 CPU 数目，就会出现 CPU 瓶颈了。这个也和 top 的负载有关系，一般负载超过了 3 就比较高了，超过了 5 就很高，超过了 10 就不正常了，服务器的状态很危险。top 的负载类似每秒的运行队列，如果运行队列过大，则表示 CPU 很繁忙，一般会造成 CPU 使用率很高。
- b，表示阻塞的进程。
- swpd，虚拟内存已使用的大小，如果大于 0，则表示机器物理内存不足了，如果不是程序内存泄露的原因，那么该升级内存或者把耗内存的任务迁移到其他机器。
- free，空闲的物理内存的大小，笔者的机器内存总共 8GB，剩余 3415MB。
- buff，Linux/UNIX 系统是用来存储的，目录里面有什么内容，权限等的缓存，笔者机器大概占用 300 多 MB。
- cache，cache 直接用来记忆我们打开的文件，给文件做缓冲，笔者机器大概占用 300 多 MB（这里是 Linux/UNIX 的聪明之处，把空闲的物理内存的一部分拿来做文件和目录的缓存，是为了提高程序执行的性能，当程序使用内存时，buffer/cached 会很快地被使用）。
- si，每秒从磁盘读入虚拟内存的大小，如果这个值大于 0，则表示物理内存不够用或者内存泄漏了，要查找耗内存进程并解决问题。

- so, 每秒虚拟内存写入磁盘的大小, 如果这个值大于 0, 处理方式同上。
- bi, 块设备每秒接收的块数量, 这里的块设备是指系统上所有的磁盘和其他块设备, 默认块大小是 1024byte, 笔者机器上没什么 I/O 操作, 所以一直是 0, 但是笔者曾在处理复制大量数据 (2~3TB) 的机器上看过可以达到 140000/s, 磁盘写入速度差不多为 140MB/S。
- bo, 块设备每秒发送的块数量。例如, 我们读取文件, bo 就要大于 0。bi 和 bo 一般都要接近 0, 不然就是 I/O 过于频繁, 需要调整。
- in, 每秒 CPU 的中断次数, 包括时间中断。
- cs, 每秒上下文切换次数。例如, 我们调用系统函数, 就要进行上下文切换、线程的切换, 这个值要越小越好, 太大了, 就要考虑调低线程或者进程的数目。例如, 在 Apache 和 Nginx 这种 Web 服务器中, 我们一般做性能测试时会进行几千并发甚至几万并发的测试, 选择 Web 服务器的进程可以由进程或者线程的峰值一直下调、压测, 直到“cs”到一个比较小的值, 这个进程和线程数就是比较合适的值了。系统调用也是, 每次调用系统函数, 我们的代码就会进入内核空间, 导致上下文切换, 这个是很耗资源的, 也要尽量避免频繁调用系统函数。上下文切换次数过多表示 CPU 大部分浪费在上下文切换上了, 导致 CPU 干正经事的时间少了, CPU 没有充分利用, 是不可取的。
- us, 用户 CPU 时间, 笔者曾经在一个做加密解密很频繁的服务器上, 看到 us 接近 100, r 运行队列达到 80 (机器在做压力测试, 性能表现不佳)。
- sy, 系统 CPU 时间, 如果太高, 则表示系统调用时间长, 例如, I/O 操作频繁。
- id, 空闲 CPU 时间, 一般来说, $id + us + sy = 100$, 一般笔者认为 id 是空闲 CPU 使用率, us 是用户 CPU 使用率, sy 是系统 CPU 使用率。
- wt, 等待 I/O CPU 时间。

10.3 运维监控

监控系统是整个运维环节, 乃至整个产品生命周期中最重要的一环, 事前及时预警发现故障, 事后提供翔实的数据用于追查定位问题。

监控系统业界有很多杰出的开源监控系统。下面就来分别介绍两款常用的运维监控软件。

10.3.1 Zabbix

Zabbix 是一个基于 Web 界面并提供分布式系统监视和网络监视功能的企业级开源解决方案。

Zabbix 主要的组件有两个：zabbix-server 和 zabbix-agent。支持的监控协议有 ICMP、IPMI、SNMP、HTTP 和 Zabbix 协议（Zabbix 协议是最常使用的协议，用来监控各个被监控端）。

收集的数据存放在数据库中，数据库支持 MySQL、Oracle 等。

第三个组件：zabbix web gui，这个接口提供 Web 页面来监控和管理各个被监控端。

第四个组件：zabbix proxy（实现分布式监控专用组件非必要组件，被监控服务器超过一千以上可以使用此组件）。

Zabbix 的运行架构如图 10-4 所示。

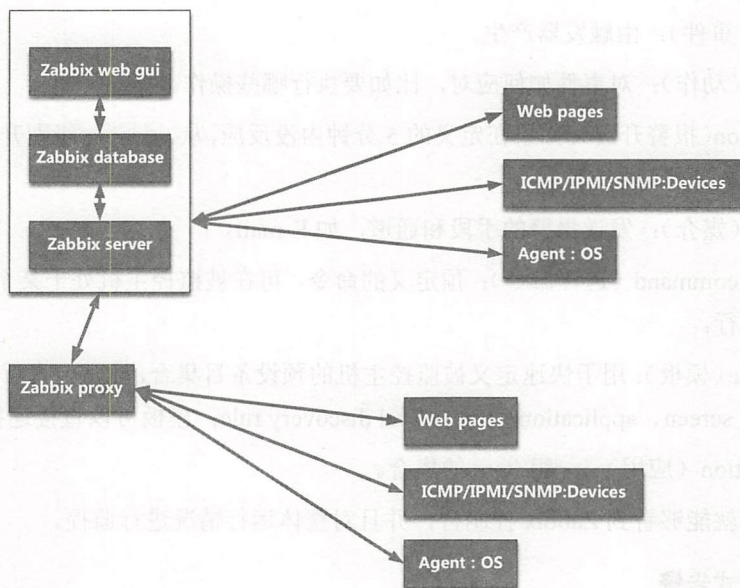


图 10-4 Zabbix 的运行架构

Zabbix 易于管理和配置，能生成比较漂亮的数据图，其自动发现功能大大减轻了日常管理的工作量，丰富的数据采集方式和 API 接口可以让用户灵活进行数据采集，而分布式系统架构可以支持监控更多的设备。理论上，通过 Zabbix 提供的插件式架构，可以满足企业的任何需求。

优点：

- 支持多平台的企业级分布式开源监控软件；
- 安装部署简单，多种数据采集插件灵活集成；
- 功能强大，可实现复杂多条件告警；
- 自带画图功能，得到的数据可以绘成图形；
- 提供多种 API 接口，支持调用脚本；

- 出现问题时可自动远程执行命令（需对 agent 设置执行权限）。

Zabbix 一些常用术语如下。

- host（主机）：要监控的网络设备，可由 IP 或 DNS 名称指定。
- host group（主机组）：主机的逻辑容器，可以包含主机和模板，但同一个组内的主机和模板不能互相连接；主机组通常在给用户或用户组指派监控权限时使用。
- item（监控项）：这个从名字上可以理解，具体要监控哪些指标由它定义。
- trigger（触发器）：超过了定义的合理范围，触发器就会报警。
- event（事件）：由触发器产生。
- action（动作）：对事件如何应对，比如要执行哪些操作。
- escalation（报警升级）：如果在定义的 5 分钟内没反应，从 warning 级别升到 high 级别，提醒别人要尽快处理。
- media（媒介）：发送报警的手段和通道，如 E-mail。
- remote command（远程命令）：预定义的命令，可在被监控主机处于某个特定条件下时自动执行。
- template（模板）：用于快速定义被监控主机的预设条目集合，通常包含了 item、trigger、graph、screen、application 及 low-level discovery rule；模板可以直接连接至单个主机。
- application（应用）：一组 item 的集合。

安装完成后就能够看到 Zabbix 控制台，并且对整体运行情况进行监控。

通过微信方式告警

Zabbix 可以通过多种方式把告警信息发送给指定人，常用的有邮件、短信等报警方式，但是越来越多的企业开始使用 Zabbix 结合微信作为主要的告警方式，这样可以及时有效地把告警信息推送给接收人，方便告警的及时处理。

需要准备：

- 微信企业号，地址为 <https://work.weixin.qq.com/>。
- 企业号已经被部门成员关注。
- 企业号有一个可以发送消息的应用，授权管理员可以使用应用给成员发送消息。
- 需要记录 CorpID 和 Secret。然后单击通讯录，添加一个组，或者添加一个成员。成员可以使用微信邀请或者短信邀请。

修改 Zabbix.conf:

```
grep alertscripts /etc/zabbix/zabbix_server.conf
AlertScriptsPath=/usr/lib/zabbix/alertscripts
```

设置 Python 脚本:

安装 simplejson

```
wget https://pypi.python.org/packages/f0/07/26b519e6ebb03c2a74989f7571e-
6ae6b82e9d7d81b8de6fdbfc643c7b58/simplejson-3.8.2.tar.gz
tar zxvf simplejson-3.8.2.tar.gz && cd simplejson-3.8.2
python setup.py build
python setup.py install
```

下载 wechat.py 脚本:

```
git clone https://github.com/X-Mars/Zabbix-Alert-WeChat.git
cp Zabbix-Alert-WeChat/wechat.py /usr/lib/zabbix/alertscripts/
cd /usr/lib/zabbix/alertscripts/
chmod +x wechat.py && chown zabbix:zabbix wechat.py
```

然后修改 wechat.py:

```
#!/usr/bin/python
#_*_coding:utf-8_*_

import urllib,urllib2
import json
import sys
import simplejson

reload(sys)
sys.setdefaultencoding('utf-8')

def gettoken(corpid,corpsecret):
    gettoken_url = 'https://qyapi.weixin.qq.com/cgi-bin/gettoken?corpid='
+ corpid + '&corpsecret=' + corpsecret
    print gettoken_url
    try:
```

```

    token_file = urllib2.urlopen(gettoken_url)
except urllib2.HTTPError as e:
    print e.code
    print e.read().decode("utf8")
    sys.exit()
token_data = token_file.read().decode('utf-8')
token_json = json.loads(token_data)
token_json.keys()
token = token_json['access_token']
return token

def senddata(access_token,user,subject,content):

    send_url = 'https://qyapi.weixin.qq.com/cgi-bin/message/send?access_
token=' + access_token
    send_values = {
        "touser":user,      #企业号中的用户账号，在 Zabbix 用户 Media 中配置，如果配置
        #不正常，则按部门发送
        "toparty":"2",      #企业号中的部门 id
        "msgtype":"text",    #消息类型
        "agentid":"2",       #企业号中的应用 id
        "text":{
            "content":subject + '\n' + content
        },
        "safe":"0"
    }
    # send_data = json.dumps(send_values, ensure_ascii=False)
    send_data = simplejson.dumps(send_values, ensure_ascii=False).encode('utf-8')
    send_request = urllib2.Request(send_url, send_data)
    response = json.loads(urllib2.urlopen(send_request).read())
    print str(response)

if __name__ == '__main__':
    user = str(sys.argv[1])    #Zabbix 传过来的第一个参数
    subject = str(sys.argv[2]) #Zabbix 传过来的第二个参数
    content = str(sys.argv[3]) #Zabbix 传过来的第三个参数

    corpid = 'corpid'         #CorpID 是企业号的标识

```



```
corpsecret = 'corpsecretsecret' #corpsecretSecret 是管理组凭证密钥
accesstoken = gettoken(corpid,corpsecret)
senddata(accesstoken,user,subject,content)
```

执行脚本，就可以在微信中收到相应的告警信息。然后在 Zabbix 中进行相应的设置就可以实现微信告警。

10.3.2 Open-Falcon

Open-falcon 是小米运维团队从互联网公司的需求出发，根据多年的运维经验，结合 SRE、SA、DevOps 的使用经验和反馈，开发的一套面向互联网的企业级开源监控产品。

开源地址：<https://github.com/XiaoMi/open-falcon>。

此项目目前活跃度不高，已经很久没有更新了，可能是因为相对比较成熟。

优点

- 强大灵活的数据采集：自动发现，支持 falcon-agent、snmp、用户主动 push、用户自定义插件、opentsdb data model like (timestamp、endpoint、metric、key-value tags)。
- 水平扩展能力：支持每个周期上亿次的数据采集、告警判定、历史数据存储和查询。
- 高效率的告警策略管理：高效的 portal、支持策略模板、模板继承和覆盖、多种告警方式、支持 callback 调用。
- 人性化的告警设置：最大告警次数、告警级别、告警恢复通知、告警暂停、不同时段不同阈值、支持维护周期。
- 高效率的 graph 组件：单机支撑 200 万 metric 的上报、归档、存储（周期为 1 分钟）。
- 高效的历史数据 query 组件：采用 rrdtool 的数据归档策略，秒级返回上百个 metric 一年的历史数据。
- dashboard：多维度的数据展示，用户自定义 Screen。
- 高可用：整个系统无核心单点，易运维、易部署、可水平扩展。
- 开发语言：整个系统的后端全部用 Golang 编写，portal 和 dashboard 使用 Python 编写。

架构如图 10-5 所示。

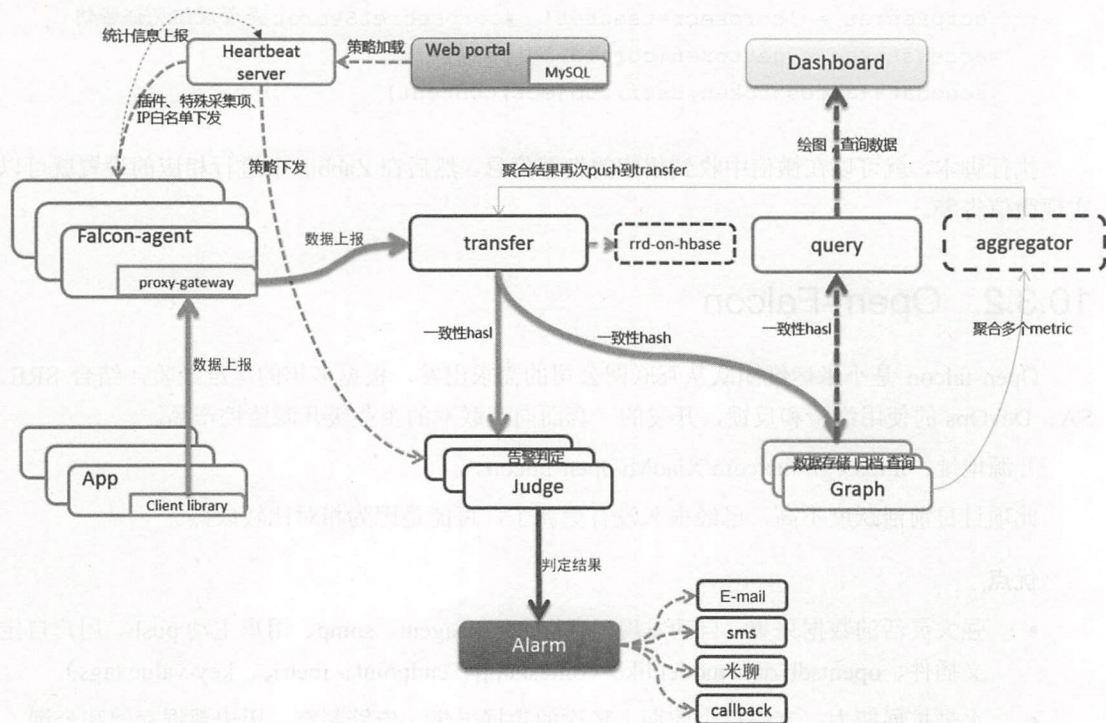


图 10-5 Open-Falcon 架构

每台服务器都安装 falcon-agent, falcon-agent 是一个 Golang 开发的 daemon 程序, 用于自发地采集单机的各种数据和指标, 这些指标包括但不限于以下几个方面, 共计 200 多项指标。

- CPU 相关;
- 磁盘相关;
- I/O;
- Load;
- 内存相关;
- 网络相关;
- 端口存活、进程存活;
- ntp offset (插件);
- 某个进程资源消耗 (插件);
- netstat、ss 等相关统计项采集;
- 机器内核配置参数。

只要安装了 falcon-agent 的机器，都会自动采集各项指标，主动上报，不需要用户在 Server 做任何配置（这和 Zabbix 有很大的不同）。这样做的好处就是用户维护方便，覆盖率高。当然这样做也会给 Server 端造成较大的压力，不过 Open-Falcon 的服务端组件单机性能足够高，同时都可以水平扩展，所以自动多采集足够多的数据，反而是一件好事情。对于 SRE 和 DevOps 来讲，事后追查问题不再是难题。

另外，falcon-agent 提供了一个 proxy-gateway，用户可以方便地通过 HTTP 接口“push”数据到本机的 Gateway，Gateway 会帮忙高效率地转发到 Server 端。

同时，它还提供了多个维度的 dashboard，可以从多个维度查看要监控的性能指标。

10.4 APM 监控

公司为什么需要分布式跟踪系统？为了支撑日益增长的庞大业务量，我们会把服务进行整合、拆分，使服务不仅能通过集群部署抵挡流量的冲击，也能根据业务在其上进行灵活的扩展。一次请求少则经过三四次服务调用完成，多则跨越几十个甚至上百个服务节点。如何动态展示服务的链路？如何分析服务链路的瓶颈并对其进行调优？如何快速进行服务链路的故障发现？这就是服务跟踪系统存在的目的和意义。

分布式跟踪系统设计要点如下。

- 对应用透明、低侵入。

这一点非常重要，如果接入成本非常高，谁都很难有耐心去针对系统做深入的研究，更别说使用其监控了。

- 低开销、高稳定。
- 可扩展。

APM = Application Performance Management，应用性能管理，对企业系统即时监控以实现应用程序性能管理和故障管理的系统化的解决方案。

现代 APM 体系基本上都是参考 Google 的 Dapper（大规模分布式系统的跟踪系统）体系来做的。通过跟踪请求的处理过程，来对应用系统在前后端处理、服务端调用的性能消耗进行跟踪。

10.4.1 Pinpoint

开源地址：<https://github.com/naver/pinpoint>。

可以看到，此款软件的活跃度比较高，一直在更新和修复 Bug。

Pinpoint 是一款对 Java 编写的大规模分布式系统的 APM 工具,有些人也喜欢称呼这类工具为调用链系统、分布式跟踪系统。我们知道,前端向后台发起一个查询请求时,后台服务可能要调用多个服务,每个服务可能又会调用其他服务,最终将结果返回,汇总到页面上。如果某个环节发生异常,工程师很难准确定位这个问题到底是由哪个服务调用造成的,Pinpoint 等相关工具的作用就是追踪每个请求的完整调用链路,收集调用链路上每个服务的性能数据,方便工程师能够快速定位问题。

Pinpoint 有以下几个特点:

- 分布式事务跟踪,跟踪跨分布式应用的消息;
- 自动检测应用拓扑,帮助你搞清楚应用的架构;
- 水平扩展以便支持大规模服务器集群;
- 提供代码级别的可见性以便轻松定位失败点和瓶颈。

架构说明:

- Pinpoint-Collector, 收集各种性能数据;
- Pinpoint-Agent, 和自己运行的应用关联起来的探针;
- Pinpoint-Web, 将收集到的数据显示成 Web 网页形式;
- HBase Storage, 将收集到的数据存到 HBase 中。

支持模块:

- JDK 6+;
- Tomcat 6/7/8, Jetty 8/9;
- Spring, Spring Boot;
- Apache HTTP Client 3.x/4.x, JDK HttpURLConnection, GoogleHttpClient, OkHttpClient;
- NingAsyncHttpClient;
- Thrift Client, Thrift Service;
- MySQL, Oracle, MSSQL, CUBRID, DBCP, PostgreSQL;
- Arcus, Memcached, Redis;
- iBatis, MyBatis;
- gson, Jackson, Json Lib;
- log4j, Logback。

Pinpoint 中的数据结构

Pinpoint 中的核心数据结构由 Span、Trace 和 TraceId 组成。

- Span: RPC (远程过程调用/remote procedure call) 跟踪的基本单元; 当一个 RPC 调用到达时指示工作已经处理完成并包含跟踪数据。为了确保代码级别的可见性, Span 将带 SpanEvent 标签的子结构作为数据结构。每个 Span 包含一个 TraceId。
- Trace: 多个 Span 的集合; 由关联的 RPC (Spans) 组成。在同一个 trace 中的 Span 共享相同的 TransactionId。Trace 通过 SpanId 和 ParentSpanId 整理为继承树结构。
- TraceId: 由 TransactionId、SpanId 和 ParentSpanId 组成的 key 的集合。TransactionId 指明消息 ID, 而 SpanId 和 ParentSpanId 表示 RPC 的父—子关系。
 - TransactionId (TxId) —— 在分布式系统间单个事务发送/接收的消息的 ID; 必须跨整个服务器集群做到全局唯一。
 - SpanId —— 当收到 RPC 消息时处理的工作的 ID; 在 RPC 请求到达节点时生成。
 - ParentSpanId (pSpanId) —— 发起 RPC 调用的父 Span 的 SpanId。如果节点是事务的起点, 则这里将没有父 Span, 对于这种情况, 使用值-1 来表示这个 Span 是事务的根 Span。

10.4.2 SkyWalking

开源地址: <https://github.com/OpenSkywalking/skywalking>。

从更新频率上来看, 此款软件更新和迭代也做得非常不错, 一直在更新和修复 Bug。

SkyWalking 3: 针对分布式系统的 APM 系统, 也被称为分布式追踪系统。

部署方式分为:

- Java 自动探针;
- 手动探针;
- 纯 Java 后端 Collector 实现, 提供 RESTful 和 GRPC 接口。兼容接受其他语言探针发送数据。

系统的使用截图如图 10-6 所示。

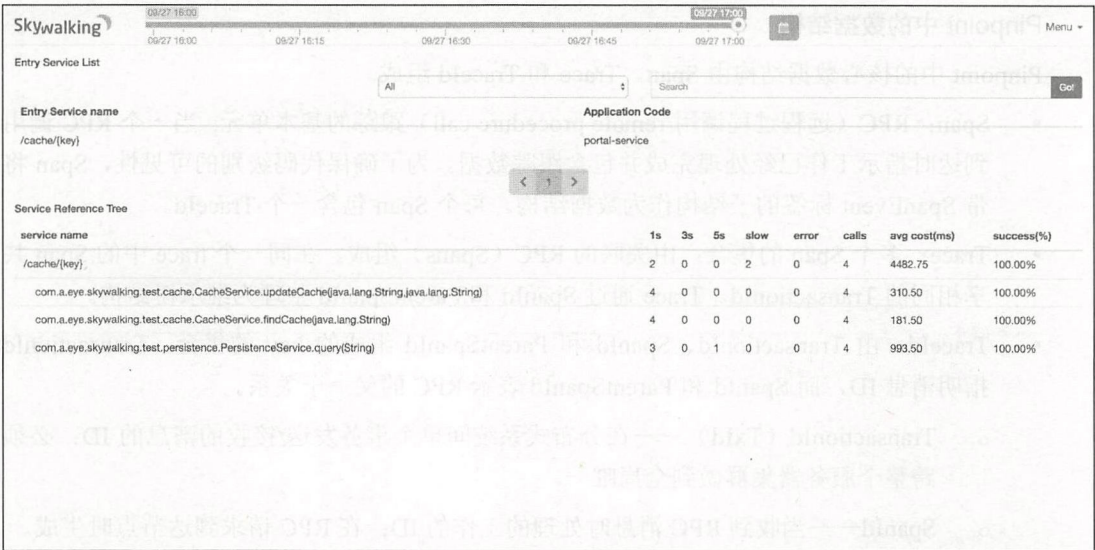


图 10-6 SkyWalking 监控界面

10.4.3 Zipkin

开源地址：<https://github.com/openzipkin/zipkin>。

从更新频率上来看，此款软件的活跃度也非常高。而且是由 Twitter 开源的，其影响力及实用性可想而知。

Zipkin 是一款开源的分布式实时数据追踪系统（Distributed Tracking System），基于 Google Dapper 的论文设计而来，其主要功能是聚集来自各个异构系统的实时监控数据。

使用场景如下。

➤ 故障快速定位

通过分析调用链，可以将一次请求的逻辑轨迹完整清晰地展示出来，只需要在业务日志中添加调用链 ID，就可以通过调用链结合业务日志快速定位错误信息。

➤ 性能分析

在调用链的各个环节分别添加调用时延，可以分析系统的性能瓶颈，进行有针对性的优化。

➤ 服务可用性

通过分析各个环节的平均时延、QPS 等信息，可以找到系统的薄弱环节，对一些模块进行调整，例如，数据冗余、链路可用等。

Zipkin 的设计主要分为三个步骤：采集、发送和落盘分析。我们来看一下 Zipkin 官网给出

的设计图，如图 10-7 所示。

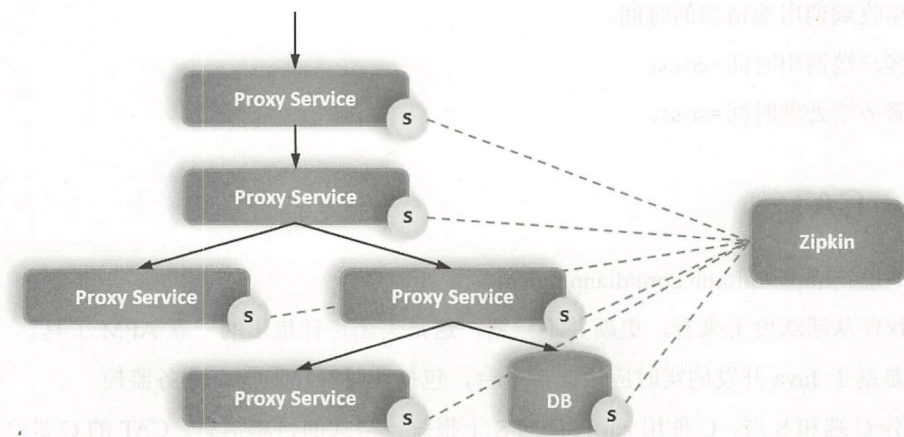


图 10-7 Zipkin 设计图

其实就是在各个点埋入 agent，也就是图 10-7 中的 S，用于采集数据。

其中几个关键概念如下。

➤ traceId

一个全局的跟踪 ID，是跟踪的入口点，根据需求来决定在哪里生成 traceId。比如一个 HTTP 请求，首先入口是 Web 应用，一般看完整的调用链，这里自然是 traceId 生成的起点，结束点在 Web 请求返回点。

➤ spanId

这是下一层的请求跟踪 ID，比如一次 RPC、一次 SQL 执行等都可以是一个 Span。一个 traceId 包含一个以上的 spanId。

➤ parentId

上一次请求跟踪 ID，用来将前后的请求串联起来。

➤ cs

客户端发起请求的时间，比如 Dubbo 调用端开始执行远程调用之前。

➤ cr

客户端收到处理完请求的时间。

➤ ss

服务端处理完逻辑的时间。

➤ sr

服务端收到调用端请求的时间。

- 客户端调用时间=cr-cs;
- 服务端处理时间=sr-ss。

10.4.4 CAT

开源地址：<https://github.com/dianping/cat>。

此款软件从活跃度上来看，更新并不频繁，这是美团点评推出的一款 APM 工具。

CAT 是基于 Java 开发的实时应用监控平台，包括实时应用监控、业务监控。

CAT 分 C 端和 S 端，C 使用 cat 接口向 S 上报统一格式的日志信息。CAT 的 C 是产生日志的地方（一般来说就是被监控的应用），相应的 S 则是接收日志、消费日志的地方（图 10-7 中的 server 节点），日志消费后生成会日志报表。

S 分为 job machine、alert machine 和 sender machine，job machine 表示可以运行定时任务的节点，alert machine 则代表可以进行告警任务的节点。定时任务将报表数据转换成图表数据；告警则基于一定规则对报表数据做筛选，剔除相应的告警数据，还有种特殊的告警用于对第三方应用做 ping 操作（2 次 ping 不通或者超时则告警），如果告警节点不同时是发送节点则只保存告警信息，如果是发送节点则发送消息提醒（微信、短信、邮箱）给相应的运维人员。

运维人员可以直接通过 S 端提供的 Web 前台功能查看报表信息和图表信息。

CAT 的监控跟传统的 APM 产品差不多，模式都是相似的，需要一个 agent 在客户端进行埋点，然后把数据发送给服务端，服务端进行解析并存储。只要你埋点足够全，那么它是可以进行全面监控的。监控到的数据会按照某种规则进行消息的合并，合并成一个 MessageTree，这个 MessageTree 会被放入 BlockingQueue 里面，这样就解决了多线程数据存储的问题。

队列会限制存储的 MessageTree 的个数，但是如果服务端挂掉，则客户端也有可能因为堆积大量的心跳而导致内存溢出。

因此数据在客户端的流程可以理解为：

Trasaction\Event→MessageTree→BlockingQueue→Netty 发出网络流

即 Transaction、Event 等消息会先合并为消息树，以消息树为单位存储在内存中（并未进行本地持久化），专门有一个 TcpSocketSender 负责向外发送数据。

再说说服务端，服务端大体上可以理解为专门有一个 TcpSocketReciever 接收数据，由于数

据在传输过程中是需要序列化的。因此接收后首先要进行 decode 处理，生成消息树。然后把消息放入 BlockingQueue，有分析器不断地来队列获取消息树进行分析，分析后按照一定的规则把报表存储到数据库，把原始数据存储到本地文件中（默认是存储到本地）。

因此数据在服务端的流程大致可以理解为：

网络流→decode 反序列化→BlockingQueue→analyzer 分析→报表存储在 DB
→原始数据存储在本地的 HDFS

树的每个节点都是一个消息（Message），消息共有 5 种具体类型，分别是 Transaction、Trace、Event、Heartbeat 和 Metric。每个消息树都有一个唯一的 messageId，且消息树之间有单父级关系。

CAT 支持的监控消息类型如下。

- **Transaction**：适合记录跨越系统边界的程序访问行为，比如远程调用、数据库调用，也适合执行时间较长的业务逻辑监控，Transaction 用来记录一段代码的执行时间和次数。
- **Event**：用来记录一件事发生的次数，比如记录系统异常，它和 Transaction 相比缺少了时间的统计，开销比 Transaction 要小。
- **Heartbeat**：表示程序内定期产生的统计信息，如 CPU%、MEM%、连接池状态、系统负载等。
- **Metric**：用于记录业务指标，指标可能包含对一个指标记录次数、记录平均值、记录总和，业务指标最低统计粒度为 1 分钟。
- **Trace**：用于记录基本的 Trace 信息，类似于 Log4j 的 info 信息，这些信息仅用于查看一些相关信息。

完整的消息树左边一列全部是时间，时间就是从程序开始到结束出现的那些序列，这里面有 URL，我们可以知道程序在这一点执行了那些关键路径，以及这个请求过程执行的所有事情，可以理解这就是一个日志的流水。我们把所有事情串起来，就知道一共做了多少事情，每一件事情都是通过时间序列串起来的。与之对应的，可以知道每一段发生的时间在哪里，每个请求的时间有多长，比较慢的是哪个，这样就知道这一段优化的方法是哪个。

监控报表包括：

- **Transaction 报表**——一段代码运行时间、次数。知道了这些，就能够对特定的请求进行定向的优化。
- **EventReport 报表**——能看到分钟级别事件的访问情况。
- **Problem 报表**——可以直观地看到需要优化的地方。可以发现异常（Java 里面抛出的

异常)、URL 访问出错、缓存访问出错、业务访问慢、数据库访问出错、服务访问出错等异常信息。

- HeartbeatReport 报表——监控系统内存、GC 及一些核心线程数。
- Cross 报表——监控整个调用环节。

10.5 Pinpoint 的安装与使用

上面列出了很多 APM 监控软件，正所谓“仁者见仁、智者见智”，选用哪一个需要根据自身公司的业务场景而定。在笔者的业务场景中选择的是 Pinpoint，下面一起来看一下 Pinpoint 的安装与使用。

10.5.1 Pinpoint 的安装

如果是测试使用，则可以直接使用 Docker 的方式进行安装，地址是：

<https://hub.docker.com/r/yous/pinpoint/>。

可以使用：

```
docker pull yous/pinpoint
```

进行安装部署。

如果是线上使用，对于 Docker 等相关技术栈不够熟悉，则可以使用实体机或者虚拟机单独部署（需要安装 JDK 1.6+、Maven 3.2+、Git）。

部署方式如下：

- （1）使用 CentOS 7，新建安装目录/pinpoint。
- （2）下载：git clone <https://github.com/naver/pinpoint.git>。

进入 Pinpoint 目录后编译：mvn install -Dmaven.test.skip=true。

- （3）下载 HBase：http://apache.mirror.cdnetworks.com/hbase/。

解压后编辑 conf/hbase-env.sh，添加 JAVA_HOME 配置：

```
export JAVA_HOME=/Library/Java/JavaVirtualMachines/jdk1.8.0_121.jdk/  
Contents/Home
```

编辑 hbase-site.xml：


```

<configuration>
  <property>
    <name>hbase.rootdir</name>
    <value>/pinpoint/tmp/hbase</value>
  </property>
  <property>
    <name>hbase.zookeeper.property.dataDir</name>
    <value>/pinpoint/tmp/zookeeper</value>
  </property>
  <property>
    <name>hbase.zookeeper.property.clientPort</name>
    <value>2182</value>
  </property>
</configuration>

```

这样配置是本地单实例模式启动，上面的配置分别表示：数据存放地点、ZooKeeper 数据存放地点、ZooKeeper 端口号（默认为 2181）。

进入 hbase/bin 目录启动 HBase：./start-hbase.sh。

用 jps 命令查看 HBase 是否启动成功，如果启动成功则会看到“HMaster”的进程。

初始化 Pinpoint 需要的表：./hbase shell hbase-create.hbase（这里的 hbase-create.hbase 在源码 pinpoint/hbase/scripts/hbase-create.hbase 中）。

访问页面测试是否成功：<http://localhost:16010/master-status>，如果成功，则在页面的 tables 标签下能看到导入的表。

也可以用命令来查看是否导入表成功，进入 HBase，输入“status 'detailed'”可以查看初始化的表。

（4）部署 Pinpoint-collector，将最新的 pinpoint-collector 的 war 放到 tomcat/webapps/下，并重命名为 ROOT.war

启动 Tomcat，配置 ROOT/WEB-INF/classes/hbase.properties：

```

hbase.client.host=localhost
hbase.client.port=2181

```

指向 ZooKeeper 的地址和端口，如果是本机，端口默认，则这里不需要更改。

完成上面的配置后，重启 Tomcat（端口为 8086）。

（5）部署 Pinpoint-web，将最新的 war 包放到另一个 Tomcat 下，并且修改 Tomcat 端口，

修改 hbase.properties，启动此 Tomcat。

(6) 部署 agent，在需要监控的机器上部署 pinpoint-agent，首先下载最新的 pinpoint-agent，然后解压缩。

配置 pinpoint.config: profiler.collector.ip=127.0.0.1，这是指 pinpoint-collector 的地址，如果是同一服务器，则不用修改。pinpoint-collector 启动后，自动就开启了 9994、9995、9996 的端口，这里默认即可。如果有端口需求，要去 pinpoint-collector 的配置文件（pinpoint-collector/webapps/ROOT/WEB-INF/classes/pinpoint-collector.properties）中修改这些端口。如果监控的是 war 包，则需要修改此 Tomcat 的 bin/catalina.sh，加入启动参数：

```
CATALINA_OPTS="$CATALINA_OPTS -javaagent:/pinpoint/pp-agent/pinpoint-bootstrap-1.6.0.jar"
CATALINA_OPTS="$CATALINA_OPTS -Dpinpoint.agentId=myapp"
CATALINA_OPTS="$CATALINA_OPTS -Dpinpoint.applicationName=MyTestPP"
```

指定 pinpoint-bootstrap-1.6.0.jar 的位置。

这里的 agentId 必须唯一，标志一个 JVM。

applicationName 表示同一种应用：同一个应用的不同实例应该使用不同的 agentId、相同的 applicationName。

(7) 如果要监控的是 Spring Boot 的 jar 包，则直接在启动命令中加启动参数：

```
nohup java -javaagent:/Users/acheron/pinpoint/pp-agent/pinpoint-bootstrap-1.6.0.jar -Dpinpoint.agentId=acheron-consumer -Dpinpoint.applicationName=crm-consumer -jar myapp.jar &
```

(8) 服务都启动后，访问 Pinpoint-web，就能够看到服务端监控的 Web 页面。

10.5.2 Pinpoint 的使用

➤ 服务器地图 (ServerMap)

通过可视化分布式系统的模块和它们之间的相互联系来理解系统拓扑。单击某个节点会展示这个模块的详情，比如它当前的状态和请求数量。

➤ 实时活动线程图表 (Realtime Active Thread Chart)

实时监控应用内部的活动线程。

➤ 请求/应答分布图表 (Request/Response Scatter Chart)，见图 10-8

可视化请求数量和应答模式来定位潜在问题。通过在图表上拖动可以选择请求查看更多的详细信息。

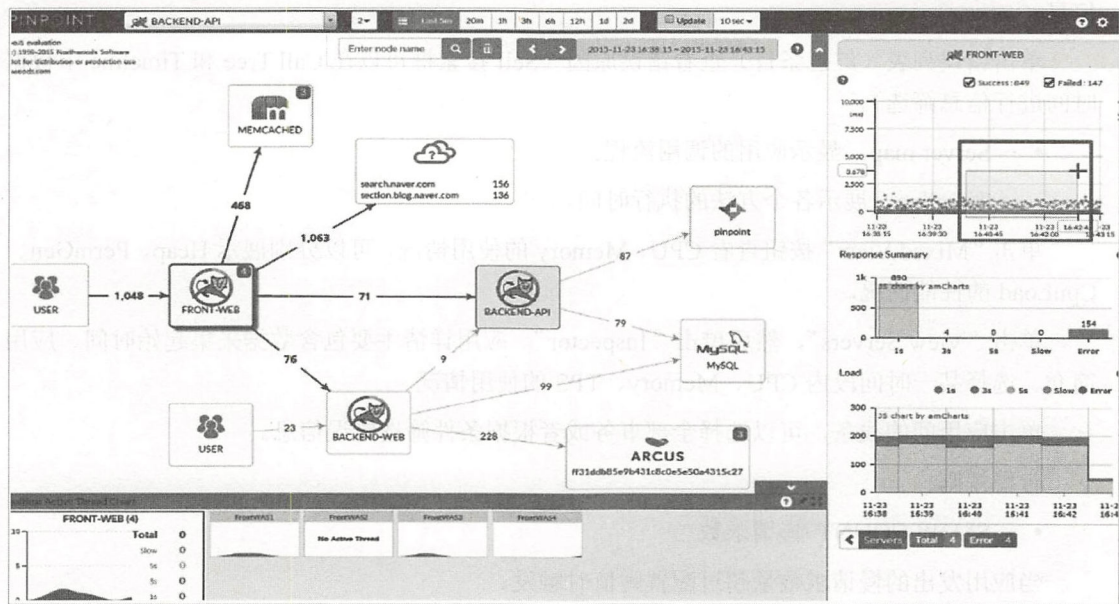


图 10-8 Pinpoint 请求应答分布界面

可以根据当前时间、最近 5 分钟、最近 20 分钟、最近 1 小时、最近 3 小时、最近 6 小时、最近 12 小时、最近 1 天，最近两天或者任意时间段的时间进行选择。

应用展示边界设置在图中顶部，此处边可以在右上角设置处统一设置。

默认是 1，就是当前应用向上或者向下跟踪的数量。

在右上角展示的是应用的成功和失败次数，以及响应时间。

如果想查看应用的调用情况，则需要在上边的框内选择然后进行拖动，这样就会弹出应用访问时间及响应时间。

可以看到以下几列信息：

- startTime，开始时间。
- path，访问路径。
- Res.(ms)，响应时间。
- Exception，异常信息。
- Agent，部署的哪个服务。

- Client IP, 客户端 IP。

红色条目代表该次请求失败, 否则代表成功。选中任意条目, 下方出现对应请求的详细信息。

单击错误列表(红色条目)查看错误原因(Self 搜索框可以在 Call Tree 和 Timeline 中根据时间进行信息筛选)。

- Server map, 显示应用的调用流程。
- Timeline, 展示各个方法的执行时间。

单击“Mixed View”按钮查看 CPU、Memory 的使用情况, 可以分别展示 Heap、PermGen、CpuLoad 的性能情况。

单击“View Servers”, 然后单击“Inspector”, 应用详情主要包含数据采集起始时间、应用简介、选择某一时间段内 CPU、Memory、TPS 的使用情况。

单击应用间的线条, 可以选择全部事务或者根据条件筛选错误信息。

告警规则:

- SLOW COUNT /慢请求数

当应用发出的慢请求数量超过配置阈值时触发。

- SLOW RATE /慢请求比例

当应用发出的慢请求百分比超过配置阈值时触发。

- ERROR COUNT /请求失败数

当应用发出的失败请求数量超过配置阈值时触发。

- ERROR RATE /请求失败率

当应用发出的失败请求百分比超过配置阈值时触发。

- TOTAL COUNT /总数量

当应用发出的所有请求数量超过配置阈值时触发。

以上规则中, 请求是当前应用发送出去的, 当前应用是请求的发起者。

以下规则中, 请求是发送给当前应用的, 当前应用是请求的接收者。

- SLOW COUNT TO CALLEE /被调用的慢请求数量

当发送给应用的慢请求数量超过配置阈值时触发。

- SLOW RATE TO CALLEE /被调用的慢请求比例

当发送给应用的慢请求百分比超过配置阈值时触发。

- ERROR COUNT TO CALLEE /被调用的请求错误数

当发送给应用的请求失败数量超过配置阈值时触发。

- ERROR RATE TO CALLEE /被调用的请求错误率

当发送给应用的请求失败百分比超过配置阈值时触发。

- TOTAL COUNT TO CALLEE /被调用的总数量

当发送给应用的所有请求数量超过配置阈值时触发。

下面两条规则和请求无关，只涉及应用的状态。

- HEAP USAGE RATE /堆内存使用率

当应用的堆内存使用率超过配置阈值时触发。

- JVM CPU USAGE RATE /JVM CPU 使用率

当应用的 CPU 使用率超过配置阈值时触发。

10.5.3 Pinpoint 实现邮件告警

Pinpoint 默认是不支持邮件告警功能的，所以需要额外添加告警功能。

下载服务器上相同的 Pinpoint 版本，地址是：<https://github.com/naver/pinpoint/releases>。

下载源代码，并且在环境变量中设置 JDK 版本，笔者使用的是 Pinpoint1.6.2。

首先配置环境变量和数据库连接，环境变量配置代码如下：

```
export JAVA_HOME=/Library/Java/JavaVirtualMachines/jdk1.8.0_91.jdk/
Contents/Home
export JAVA_6_HOME=/Library/Java/JavaVirtualMachines/jdk1.8.0_91.jdk/
Contents/Home
export JAVA_7_HOME=/Library/Java/JavaVirtualMachines/jdk1.8.0_91.jdk/
Contents/Home
export JAVA_8_HOME=/Library/Java/JavaVirtualMachines/jdk1.8.0_91.jdk/
Contents/Home
export PATH=$PATH:$JAVA_HOME/bin
export CLASSPATH=.:$JAVA_HOME/jre/lib:$JAVA_HOME/lib:$JAVA_HOME/lib/tools.jar
export PATH=$PATH:$JAVA_6_HOME/bin
export CLASSPATH=.:$JAVA_6_HOME/jre/lib:$JAVA_6_HOME/lib:$JAVA_6_HOME/
lib/tools.jar
```

```

export PATH=$PATH:$JAVA_7_HOME/bin
export CLASSPATH=.:$JAVA_7_HOME/jre/lib:$JAVA_7_HOME/lib:$JAVA_7_HOME/
lib/tools.jar
export PATH=$PATH:$JAVA_8_HOME/bin
export CLASSPATH=.:$JAVA_8_HOME/jre/lib:$JAVA_8_HOME/lib:$JAVA_8_HOME/
lib/tools.jar
export PATH=/usr/local/Cellar/maven/3.3.9/bin:$PATH

```

分别运行数据库的初始化文件 `CreateTableStatement-mysql.sql` 和 `SpringBatchJobRepository-Schema-mysql.sql`。告警信息可以使用短信和邮件的方式进行发送。需要实现 `AlarmMessageSender` 接口并将其实现注册到 `Spring` 容器中。实现代码如下：

```

public class AlarmMessageSenderImple implements AlarmMessageSender {

    private final Logger logger = LoggerFactory.getLogger(this.getClass());

    @Autowired
    private UserGroupService userGroupService;

    @Override
    public void sendSms(AlarmChecker checker, int sequenceCount) {
        List<String> receivers = userGroupService.selectPhoneNumberOfMember
(checker.getUserGroupId());

        if (receivers.size() == 0) {
            return;
        }

        for (String message : checker.getSmsMessage()) {
            logger.info("send SMS : {}", message);

            // TODO Implement logic for sending SMS
        }
    }

    @Override
    public void sendEmail(AlarmChecker checker, int sequenceCount) {
        List<String> receivers = userGroupService.selectEmailOfMember

```



```

(checker.getUserGroupId());

    if (receivers.size() == 0) {
        return;
    }

    for (String message : checker.getEmailMessage()) {
        logger.info("send email : {}", message);

        // TODO Implement logic for sending email
    }
}
}

```

注册代码如下:

```

<bean id="AlarmMessageSenderImple" class="com.navercorp.pinpoint.web.alarm.
AlarmMessageSenderImple"/>

```

然后设置 `batch.properties` 的 `batch.enable` 属性为 `true`。

配置 MySQL 指向所用的服务器的数据库。在 `applicationContext-batch-schedule.xml` 中设置调度执行的频率。

配置完成后, 执行编译命令:

```
mvn -f ./pom.xml install -Dmaven.test.skip=true
```

结果展示如图 10-9 所示。

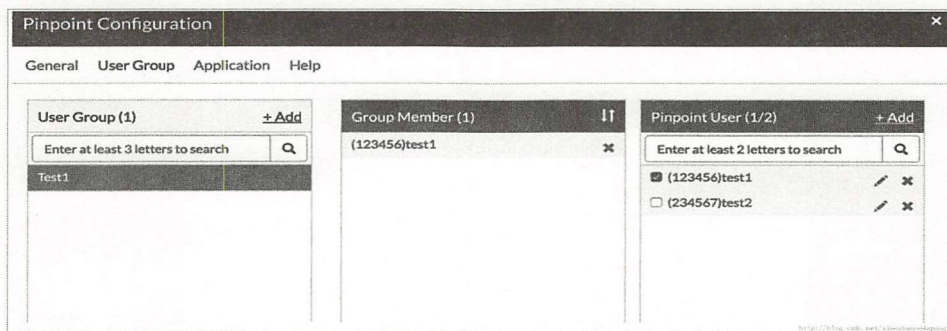


图 10-9 Pinpoint 邮件告警配置

给线上的应用加上邮件告警吧！

10.6 小结

运维的监控是整个微服务体系中非常重要的一环，通过对日志的收集，运维监控平台的搭建，以及对 APM 监控平台的使用，能够有效地发现整个软件运行周期内出现的问题，并且有针对性地进行解决，第一时间处理问题，提高开发效率。

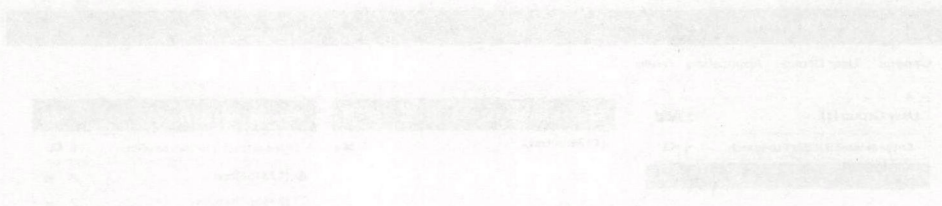


图 10-9 Prometheus 告警配置

11 chapter

第 11 章 完整示例



前面介绍了如何使用 Spring Boot、Spring Cloud、Docker 和 Netflix 等一些开源工具来构建一个微服务架构，相信通过之前章节的介绍，大家已经对上述几个概念有了一个简单的了解，并且对微服务已经有了自己的认知。

本章会介绍一个完整的示例，用于演示使用 Spring Cloud 搭建微服务的整个过程，源代码可以直接下载使用。这里介绍的是一套简单的财务管理系统，用户可以注册自己的账户，并且将每个月的收入和支出情况进行录入，系统会自动实现账户的统计和消息通知的功能。

11.1 安装 Lombok

Lombok 的官方网址：<http://projectlombok.org/>。

使用 Lombok 是需要安装的，如果不安装，则 IDE 无法解析 Lombok 注解。先在官网下载最新版本的 JAR 包。

安装方式有如下两种。

➤ 直接安装

双击下载下来的 JAR 包来安装 Lombok，选择 IDE 的安装路径，直到安装完成。

➤ 手动安装

(1) 将 `lombok.jar` 复制到 `myeclipse.ini` / `eclipse.ini` 所在的文件夹目录下。

(2) 打开 `eclipse.ini` / `myeclipse.ini`，在最后面插入以下两行并保存：

```
-Xbootclasspath/a:lombok.jar  
-javaagent:lombok.jar
```

(3) 重启 Eclipse / MyEclipse。

安装完成后，IDE 就可以正常编译使用 Lombok 了。

比较常用的注解如下。

- `@Data`：注解在类上；提供类所有属性的 `getting` 和 `setting` 方法，此外还提供了 `equals`、`canEqual`、`hashCode`、`toString` 方法。
- `@Setter`：注解在属性上；为属性提供 `setting` 方法。
- `@Getter`：注解在属性上；为属性提供 `getting` 方法。
- `@Log4j`：注解在类上；为类提供一个属性名为 `log` 的 Log4j 日志对象。
- `@NoArgsConstructor`：注解在类上；为类提供一个无参的构造方法。
- `@AllArgsConstructor`：注解在类上；为类提供一个全参的构造方法。

11.2 PiggyMetrics

下面通过 PiggyMetrics 项目的示例来一起了解微服务的架构模式和处理方式。PiggyMetrics 的 GitHub 地址为 <https://github.com/sqshq/PiggyMetrics>。

这个项目以前是一个单体应用，现在已经被构建成微服务应用。

PiggyMetrics 通过 Spring Cloud 实现微服务架构，应用被分解为账户服务（Account Service）、统计服务（Statistics Service）、通知服务（Notification Service）三个核心微服务。每个微服务都是围绕业务能力组织的可独立部署的应用程序，拥有独立的数据库并使用同步的 REST API 实现微服务与微服务之间的通信。PiggyMetrics 架构如图 11-1 所示。

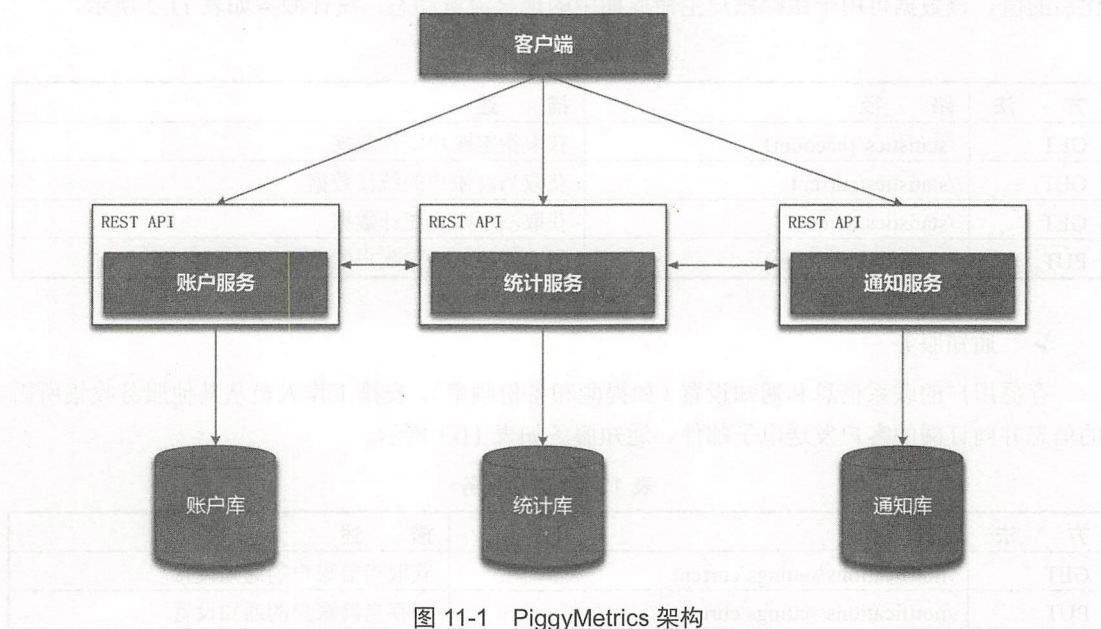


图 11-1 PiggyMetrics 架构

它们都是可以独立部署的应用，每个微服务都分别访问各自的数据库，数据库使用的是 MongoDB，因为是物理上独立部署，所以可以根据微服务的特点选择合适的数据存储方式。服务之间的通信非常简单，通过 Feign 调用同步的 REST API 进行通信，在实际的线上环境，对于小并发量的情况，这种方式是没有问题的，但是当并发量大的时候，正如在第 2 章提到的设计原则，可以使用异步的方式完成插入和更新操作。

三个微服务的接口信息如下。

➤ 账户服务

包含一般用户输入逻辑和验证：收入/开销记录、储蓄和账户设置。账户服务如表 11-1 所示。

表 11-1 账户服务

方 法	路 径	描 述
GET	/accounts/{account}	获取指定账户数据
GET	/accounts/current	获取当前账户数据
GET	/accounts/demo	获取演示账户数据（预先填入收入/开销记录等）
PUT	/accounts/current	保存当前账户数据
POST	/accounts/	注册新账户

➤ 统计服务

计算主要的统计参数，并捕获每一个账户的时间序列。数据点包含基于货币和时间段正常化后的值。该数据可用于跟踪账户生命周期中的现金流量动态。统计服务如表 11-2 所示。

表 11-2 统计服务

方 法	路 径	描 述
GET	/statistics/{account}	获取指定账户统计数据
GET	/statistics/current	获取当前账户的统计数据
GET	/statistics/demo	获取演示账户统计数据
PUT	/statistics/{account}	创建或更新指定账户的时间序列数据点

➤ 通知服务

存储用户的联系信息和通知设置（如提醒和备份频率）。安排工作人员从其他服务收集所需的信息并向订阅的客户发送电子邮件。通知服务如表 11-3 所示。

表 11-3 通知服务

方 法	路 径	描 述
GET	/notifications/settings/current	获取当前账户的通知设置
PUT	/notifications/settings/current	保存当前账户的通知设置

11.3 整体架构

除了业务服务，还有其他几个公共基础服务，这些服务都比较通用。

项目的整体架构如图 11-2 所示。

整个架构说明如下：

- 业务服务的调用关系是账户服务通过远程客户端（Feign）调用统计服务及通知服务，通过 Ribbon 实现负载均衡，并在调用过程中增加了断路器（Hystrix）的功能。

- API Gateway (Zuul) 提供对外统一的服务网关，首先从注册中心 (Eureka) 获取相应服务，再根据业务需要调用各个服务的逻辑。

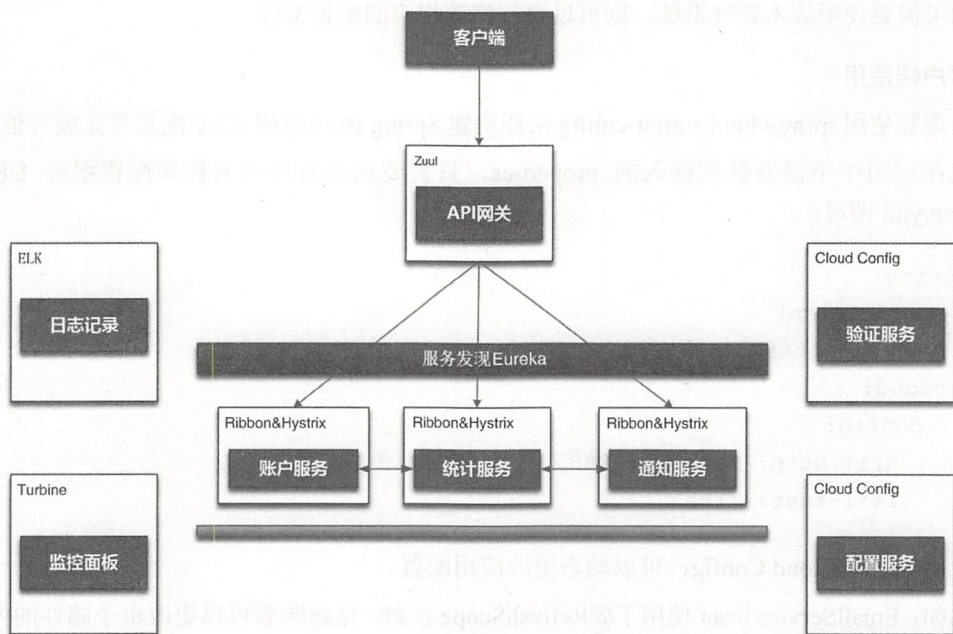


图 11-2 PiggyMetrics 整体架构

公共服务包括：

- Turbine 监控所有的微服务的运行情况，当出现问题时能够及时发现问题。
- 整个业务过程中所有服务的配置文件通过 Spring Cloud Config 来管理，即起什么端口、配置什么参数等。
- 认证机制通过 Auth service 实现，提供基本认证服务。

Spring Cloud Config、Eureka、Ribbon、Hystrix、Feign 及 Turbine 均为标准组件，与业务之间没有强关联，不涉及业务代码，仅需简单配置即可工作。所以，这些公共模块可以做成高可用，在需要调用的时候调用即可。

11.3.1 配置 Spring Cloud Config

Spring Cloud Config 是分布式系统的水平扩展集中式配置服务。它目前支持的本地存储、Git 和 Subversion 等方式存储。

在此项目中，使用了 native profile，它简单地从本地 classpath 下加载配置文件。可以在配

置服务资源中查看 `shared` 目录。现在，当通知服务请求它的配置时，配置服务将监控 `shared/notification-service.yml` 和 `shared/application.yml`，以供所有客户端应用之间共享。

如果需要使用版本管理系统，则可以自行修改相应的配置文件。

客户端使用

只需要使用 `spring-cloud-starter-config` 依赖构建 Spring Boot 应用，自动配置将完成其他工作。

现在应用中不需要任何嵌入的 `properties`，只需要提供有应用名称和配置服务 URL 的 `bootstrap.yml` 即可：

```
Spring:
  application:
    name: notification-service
  cloud:
    config:
      uri: http://config:8888
      fail-fast: true
```

使用 Spring Cloud Config，可以动态更改应用配置。

比如，`EmailService` bean 使用了 `@RefreshScope` 注解。这意味着可以更改电子邮件的内容和主题，而无须重新构建和重启通知服务应用。

首先，在配置服务器中更改必要的属性。然后，对通知服务执行刷新请求：`curl -H "Authorization: Bearer #token#" -XPOST http://127.0.0.1:8000/notifications/refresh。`

也可以使用 Webhook 来自动执行此过程，比较常用的方式是使用 Jenkins 的 job 中构建触发器。

注意：

(1) 动态刷新存在一些限制。`@RefreshScope` 不能和 `@Configuraion` 类一同工作，并且不会作用于 `@Scheduled` 方法。

(2) `fail-fast` 属性意味着如果 Spring Boot 应用无法连接到配置服务，则会立即启动失败。当一起启动所有应用时，这非常有用。

11.3.2 授权服务

授权完全由单独的服务器完成，它为后端资源服务授予 OAuth 2 tokens 令牌。使用密码凭据（类型为 Password credentials）实现用户授权，使用客户端凭证 Client Credentials 实现微服务授权。

Spring Cloud Security 提供了方便的注释和自动配置,使得服务器端和客户端都很容易实现。如果控制器不想被外部访问,则可以指定访问的范围,代码如下所示。

```
@PreAuthorize("#oauth2.hasScope('server')")
@RequestMapping(value = "accounts/{name}", method = RequestMethod.GET)
public List<DataPoint> getStatisticsByAccountName(@PathVariable String
name) {
    return statisticsService.findByAccountName(name);
}
```

11.3.3 API 网关

可以看到,有三个核心服务向客户端暴露外部 API。在现实系统中,这个数量可以非常快地增长,同时整个系统将变得非常复杂。实际上,一个复杂页面的渲染可能涉及数百个服务。

理论上,客户端可以直接向每个微服务发送请求。但这种方式是非常不安全的,需要知道所有端点的地址,分别对每一段信息执行 HTTP 请求,将结果合并到客户端。API 网关是系统的单个入口点,用于将请求路由到适当的后端服务,或者调用多个后端服务并聚合结果来处理请求。关于为何使用网关可以参考 Spring Cloud 中网关的使用一节。

我们可以用一个 `@EnabledZuulProxy` 注解来启用它。在这个项目中,使用 Zuul 存储静态内容,并将请求路由到适当的微服务。以下是一个简单的基于前缀 (prefix-based) 路由的通知服务配置:

```
zuul:
  routes:
    notification-service:
      path: /notifications/**
      serviceId: notification-service
      stripPrefix: false
```

这意味着所有以 `/notification` 开头的请求将被路由到通知服务。可以看到,里面没有硬编码的地址。Zuul 使用服务发现机制来定位通知服务实例以及断路器和负载均衡器。

11.3.4 服务发现

服务发现的关键部分是服务注册。当客户端需要负责确定可以用的服务实例的位置和跨平台的负载均衡请求时,Eureka 就是客户端发现模式的一个很好的例子。使用 Spring Boot,通过

Spring-cloud-starter-eureka-server 依赖、@EnabledEurekaServer 注解和简单的配置属性可以轻松构建 Eureka 注册中心。

使用 @EnabledDiscoveryClient 注解和带有应用名称的 bootstrap.yml 来启用客户端支持：

```
Spring:
  application:
    name: notification-service
```

在应用启动时，它向 Eureka 服务器注册并提供元数据，如主机和端口、健康指示器 URL、主页等。Eureka 接收来自从属于某服务的每个实例的心跳消息。如果心跳失败超过配置的时间，则该实例将从注册表中删除。

此外，Eureka 还提供了一个简单的界面，可以通过它来跟踪运行中的服务和可用实例的数量。

通过输入以下地址就可以看到所有正在运行的可用实例：

```
http://localhost:8761
```

11.3.5 负载均衡器、断路器和 HTTP 客户端

Ribbon

Ribbon 是一个客户端负载均衡器，可以很好地控制 HTTP 和 TCP 客户端的行为。它与 Spring Cloud 和服务发现是集成在一起的，可开箱即用。Eureka 客户端提供了可用服务器的动态列表，因此 Ribbon 可以在它们之间进行负载均衡。

此项目中并没有显式地定义 Ribbon 的使用，但是在 Zuul、Feign 等组件中隐式地使用了 Ribbon，我们在实际的业务开发中，也不需要刻意定义 Ribbon。

Hystrix

Hystrix 是断路器模式的一种实现，它可以通过网络访问依赖来控制延迟和故障。核心思想是在具有大量微服务的分布式环境中停止发生的雪崩效应。这有助于快速失败并尽快恢复，微服务中的自我修复在系统设计中是非常重要的。

除了断路器控制，在使用 Hystrix 时，可以添加一个备用方法，在主命令失败的情况下，该方法将被调用以获取默认值。

此外，Hystrix 生成每个命令的执行结果和延迟的度量，我们可以用它来监视系统的行为。

Feign

Feign 是一个声明式 HTTP 客户端，能与 Ribbon 和 Hystrix 无缝集成。实际上，通过一个 Spring-cloud-starter-feign 依赖和@EnabledFeignClients 注解，可以使用一整套负载均衡器、断路器和 HTTP 客户端，并附带一个合理的默认配置。此项目中多次使用了 Feign。

以下是账户服务的示例：

```
@FeignClient(name = "statistics-service")
public interface StatisticsServiceClient {
    @RequestMapping(method = RequestMethod.PUT, value = "/statistics/
{accountName}", consumes = MediaType.APPLICATION_JSON_UTF8_VALUE)
    void updateStatistics(@PathVariable("accountName") String accountName,
Account account);
}
```

需要的只是一个接口，可以在 Spring MVC 控制器和 Feign 方法之间共享@RequestMapping 部分。

11.3.6 监控仪表盘

项目中统一定义了熔断策略，配置信息如下：

```
hystrix:
  command:
    default:
      execution:
        isolation:
          thread:
            timeoutInMilliseconds: 10000    ## 10000ms 超时限制
```

Hystrix 的每一个微服务都通过 Spring Cloud Bus 将指标推送到 Turbine，只需要在客户端添加如下代码即可实现：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-netflix-hystrix-stream</artifactId>
</dependency>
```

让我们看看系统行为在负载下：账户服务调用统计服务，在一个变化的模拟延迟下的响应，响应超时阈值设置为 1 秒，如图 11-3 所示。

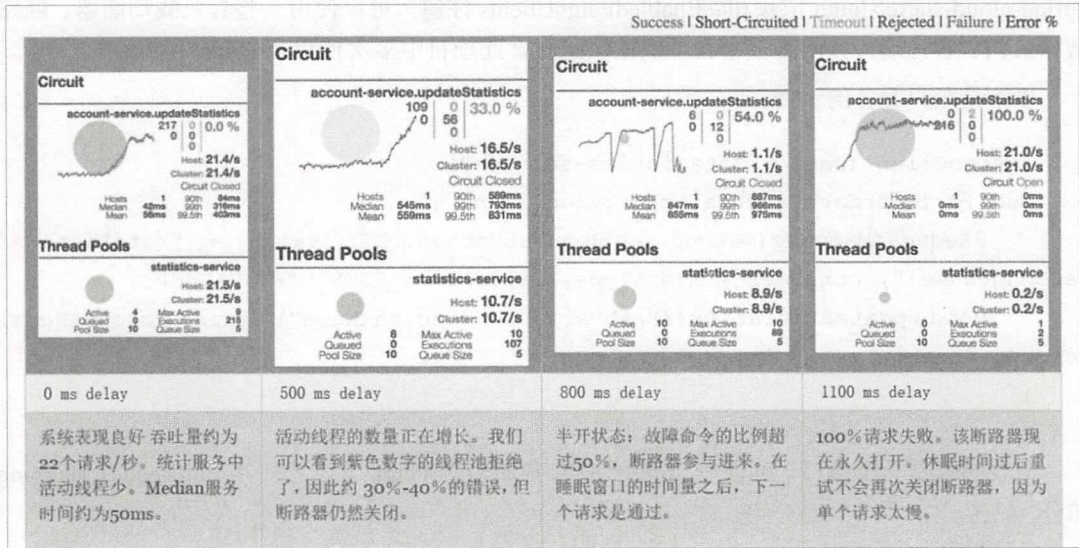


图 11-3 PiggyMetrics 监控仪表盘

11.3.7 日志分析

集中式日志记录在尝试查找分布式环境中的问题时非常有用。Elasticsearch、Logstash 和 Kibana 技术栈可搜索并分析日志、利用率和网络活动数据。

11.4 安装和运行

11.4.1 配置 Maven 并导入工程

因为在国内使用的原因，这里配置为阿里云的仓库地址。

在当前登录用户目录的 .m 目录下面的 settings.xml 文件中，在<mirrors></mirrors>段落里面有如下内容：

```
<mirror>
  <id>alimaven</id>
  <name>alimaven</name>
```



```
<url>http://maven.aliyun.com/nexus/content/groups/public/</url>
<mirrorOf>central</mirrorOf>
</mirror>
```

打开 STS 并使用 Maven 导入工程。

工程导入后的依赖关系如下：

```
<modules>
  <module>config</module>
  <module>monitoring</module>
  <module>registry</module>
  <module>gateway</module>
  <module>auth-service</module>
  <module>account-service</module>
  <module>statistics-service</module>
  <module>notification-service</module>
</modules>
```

执行 `mvn package -DskipTests` 进行打包，执行后在每一个项目的 `target` 下有两个 jar 包。

微服务工程的目录结构为：

Client	Feign 客户端
Controller	控制层
Domain	实体层
Responsity	DAO 层
Service	服务层

每个项目下都有 `Dockerfile` 文件和 `src/main/resources/bootstrap.yml` 文件，以 `account-service` 为例。

`Dockerfile` 文件的内容如下：

```
FROM java:8-jre
MAINTAINER Alexander Lukyanchikov <sqshq@sqshq.com>

ADD ./target/account-service.jar /app/
CMD ["java", "-Xmx200m", "-jar", "/app/account-service.jar"]

EXPOSE 6000
```

- java:8-jre 是指 Docker 上官方提供的 Java 镜像，版本号是 8，也就是 JDK1.8。
- MAINTAINER 指作者信息。
- ADD 指将应用 jar 包复制到/app/中。
- CMD 表示容器默认执行的命令。

配置中心 config 的代码如下：

```
Spring:
  cloud:
    config:
      server:
        native:
          search-locations: classpath:/shared #指向 shared 文件夹
      profiles:
        active: native

server:
  port: 8888 #端口

security:
  user:
    password: ${CONFIG_SERVICE_PASSWORD}
```

11.4.2 安装

确保环境里安装了最新版本的 Docker 和 Docker Compose。

设置环境变量：

```
export CONFIG_SERVICE_PASSWORD=root
export NOTIFICATION_SERVICE_PASSWORD=root
export STATISTICS_SERVICE_PASSWORD=root
export ACCOUNT_SERVICE_PASSWORD=root
export MONGODB_PASSWORD=root ## 必填，其他变量可不设置
```

进入 PiggyMetrics 工程目录，确保本目录下有 docker-compose.yml 文件，然后执行：

```
docker-compose -f docker-compose.yml up -d
```


开始从 Docker hub 中拉取镜像:

```
Pulling statistics-mongodb (sqshq/piggymetrics-mongodb:latest)...
latest: Pulling from sqshq/piggymetrics-mongodb
5233d9aed181: Pull complete
5bbfc055e8fb: Pull complete
```

构建完成后, 执行 `docker ps`, 能够看到各个模块的运行情况。

笔者本地使用的测试机的 IP 是 `xx.xx.xx.xx`, 输入:

```
http:// xx.xx.xx.xx:8761/
```

能够看到 Eureka 上的各个注册的微服务模块。

ACCOUNT-SERVICE、ACCOUNT-SERVICE、GATEWAY、NOTIFICATION-SERVICE、STATISTICS-SERVICE 这五个服务会注册在服务注册中心。如果有新的服务, 也可以通过注册中心进行注册。

输入 `http:// xx.xx.xx.xx:15672/`, 打开 RabbitMQ 管理界面, 输入用户名和密码(都使用 `guest`), 可以看到如图 11-4 所示的界面。

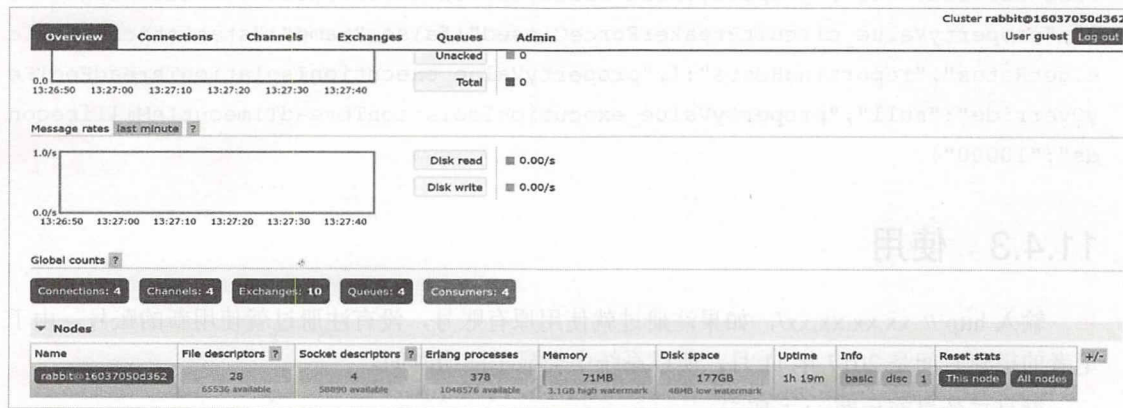


图 11-4 RabbitMQ 监控面板

输入 `http:// xx.xx.xx.xx:9000/hystrix`, 在地址栏输入数据地址 `http://xx.xx.xx.xx:8989:`

data:

```
{"rollingCountFallbackFailure":0,"rollingCountFallbackSuccess":0,
```

```
"propertyValue_circuitBreakerRequestVolumeThreshold":"20","propertyValue_circuitBreakerForceOpen":false,"propertyValue_metricsRollingStatisticalWindowInMilliseconds":"10000","latencyTotal_mean":0,"type":"HystrixCommand","rollingCountResponsesFromCache":0,"TypeAndName":"TypeAndName=>HystrixCommand_statistics-service.getRates","rollingCountTimeout":0,"propertyValue_executionIsolationStrategy":"THREAD","instanceId":"statistics-service:7000","rollingCountFailure":0,"rollingCountExceptionsThrown":0,"latencyExecute_mean":0,"isCircuitBreakerOpen":false,"errorCount":0,"group":"rates-client","rollingCountSemaphoreRejected":0,"latencyTotal":{"0":0,"25":0,"50":0,"75":0,"90":0,"95":0,"99":0,"99.5":0,"100":0},"requestCount":0,"rollingCountCollapsedRequests":0,"rollingCountShortCircuited":0,"latencyExecute":{"0":0,"25":0,"50":0,"75":0,"90":0,"95":0,"99":0,"99.5":0,"100":0},"propertyValue_circuitBreakerSleepWindowInMilliseconds":"5000","currentConcurrentExecutionCount":0,"propertyValue_executionIsolationSemaphoreMaxConcurrentRequests":"10","errorPercentage":0,"rollingCountThreadPoolRejected":0,"propertyValue_circuitBreakerEnabled":true,"propertyValue_executionIsolationThreadInterruptOnTimeout":true,"propertyValue_e_requestCacheEnabled":true,"rollingCountFallbackRejection":0,"propertyValue_requestLogEnabled":true,"rollingCountSuccess":0,"propertyValue_fallbackIsolationSemaphoreMaxConcurrentRequests":"10","InstanceKey":"InstanceKey=>statistics-service:7000","propertyValue_circuitBreakerErrorThresholdPercentage":"50","propertyValue_circuitBreakerForceClosed":false,"name":"statistics-service.getRates","reportingHosts":1,"propertyValue_executionIsolationThreadPoolKeyOverride":"null","propertyValue_executionIsolationThreadTimeoutInMilliseconds":"10000"}
```

11.4.3 使用

输入 `http://xx.xx.xx.xx/`，如果注册过就使用原有账号，没有注册过就使用新的账号。由于笔者的搭建时间是 2017 年 11 月，所以系统中会显示注册时间。

登录后的界面如图 11-5 所示。

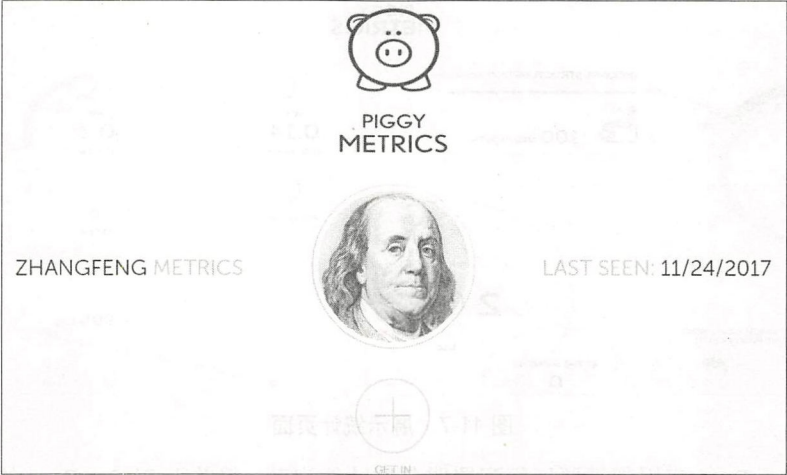


图 11-5 登录后的展示页面

左半部分是登录用户信息，右半部分是最后一次登录的时间。

单击 GET IN，将出现如图 11-6 所示的界面。

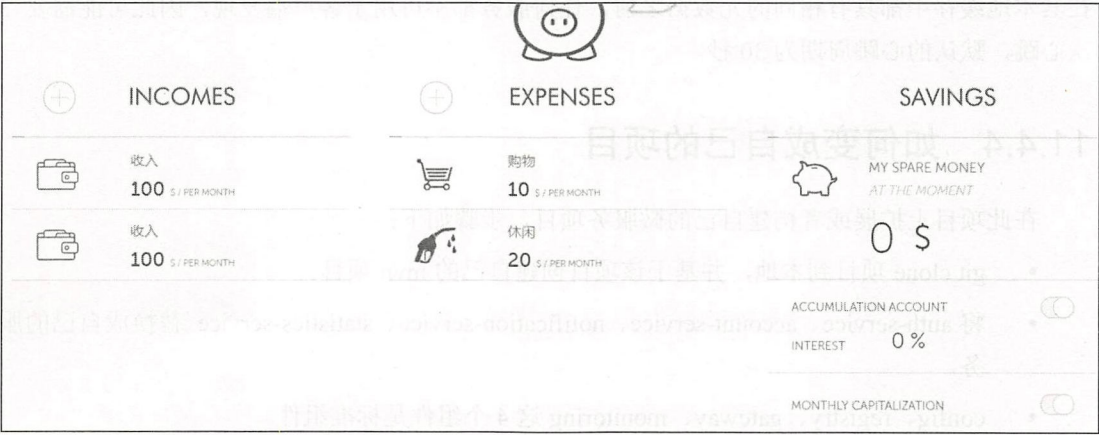


图 11-6 PiggyMetrics 展示用户的收入和支出信息

界面上显示此用户的收入、支出信息。

添加入收入和支出记录，保存后展示如图 11-7 所示的界面。

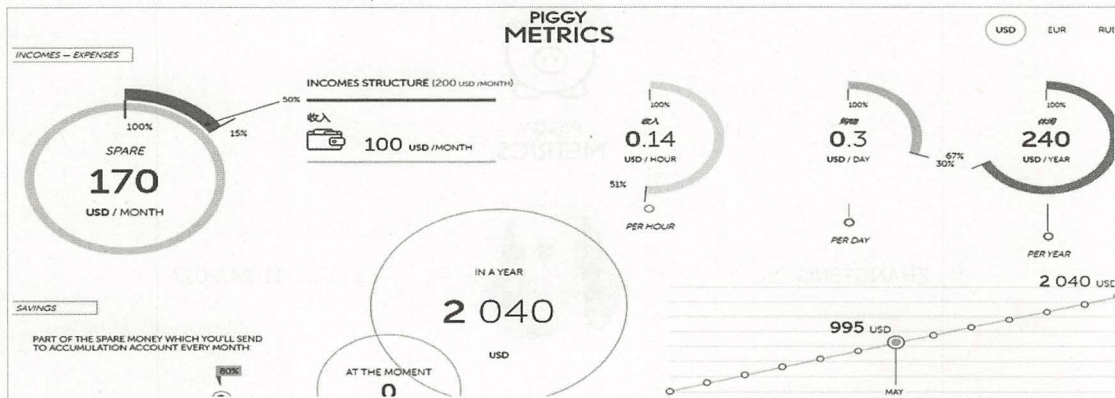


图 11-7 展示统计页面

所有 Spring Boot 应用都需要运行配置服务器才能启动。得益于 Spring Boot 的 fail-fast 属性和 docker-compsoe 的 restart:always 选项，我们可以同时启动所有容器。这意味着所有依赖的容器将尝试重新启动，直到配置服务器启动运行为止。

此外，服务发现机制在所有应用启动后需要等待一段时间。实例、Eureka 服务器和客户端在其本地缓存中都具有相同的元数据之前，任何服务都不可用于客户端发现，因此可能需要 3 次心跳，默认的心跳周期为 30 秒。

11.4.4 如何变成自己的项目

在此项目上扩展或者构建自己的微服务项目，步骤如下：

- git clone 项目到本地，并基于该项目创建自己的 mvn 项目。
- 将 auth-service、account-service、notification-service、statistics-service 替换成自己的服务。
- config、registry、gateway、monitoring 这 4 个组件是标准组件。

在实际业务的开发中，在 GatewayApplication.java 中用具体业务替换相应的服务即可。

```
@EnableZuulProxy      ## 增加 Zuul Proxy 代理功能
public class GatewayApplication {
    public static void main(String[] args) {
        SpringApplication.run(GatewayApplication.class, args);
    }
}
```


在 resources 目录下增加 static 存放静态资源（如 HTML、CSS、Images 等）。

在 Zuul 的配置文件 gateway.yml 中增加代理服务的配置：

```
zuul:
  ignoredServices: '*'
  host:
    connect-timeout-millis: 20000          ## 超时时间
    socket-timeout-millis: 20000
  routes:
    auth-service:                          ## 认证服务
      path: /uaa/**                        ## 匹配路径
      url: http://auth-service:5000        ## 服务路径（HTTP 方式）
      stripPrefix: false                   ## 是否包括前缀
      sensitiveHeaders:
    account-service:
      path: /accounts/**
      serviceId: account-service           ## 通过服务 ID 动态查找
      stripPrefix: false
      sensitiveHeaders:
    statistics-service:
      path: /statistics/**
      serviceId: statistics-service
      stripPrefix: false
      sensitiveHeaders:
    notification-service:
      path: /notifications/**
      serviceId: notification-service
      stripPrefix: false
      sensitiveHeaders:
```

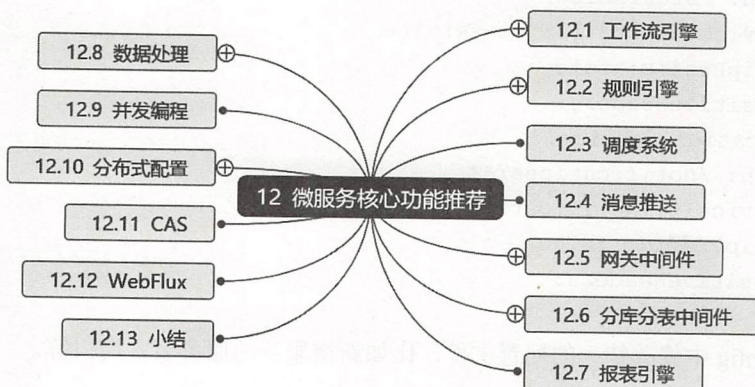
- 在 config 中修改统一的配置文件，比如新增服务的服务名、端口等。
- 通过 mvn 构建后生成镜像。
- 运行所有的镜像。

11.5 小结

PiggyMetrics 是一个供个人处理财务的解决方案，也是一个学习微服务非常好的架构，本章介绍了此工程的使用方法，可以在此框架的基础上进行扩展，实现微服务的开发。

12 chapter

第 12 章 微服务核心功能推荐



构建微服务，除了正常的业务功能，还需要其他很多的辅助功能来帮我们一起完成整体微服务的业务流程。下面我们看一下有哪些构建微服务能使用到的开源组件。

12.1 workflow引擎

微服务中，workflow引擎起着非常重要的作用，那么workflow在微服务中到底扮演着什么样的角色呢？这就要从什么是workflow说起了。workflow（workflow）就是工作流程的计算模型，即将工作流程中的工作如何前后组织在一起的逻辑和规则在计算机中以恰当的模式进行表示并对其实施计算。它主要解决的是“使在多个参与者之间按照某种预定义的规则传递文档、信息或任务的过程自动进行，从而实现某个预期的业务目标，或者促使此目标的实现”。

12.1.1 Activiti

Activiti 是一个开源的workflow引擎，它实现了 BPMN 2.0 规范，可以发布设计好的流程定义，并通过 API 进行流程调度。

Activiti 作为一个遵从 Apache 许可的workflow和业务流程管理开源平台，其核心是基于 Java 的超快速、超稳定的 BPMN 2.0 流程引擎，强调流程服务的可嵌入性和可扩展性，同时更加强调面向业务人员。

Activiti 流程引擎重点关注系统开发的易用性和轻量性。每一项 BPM 业务功能 Activiti 流程引擎都以服务的形式提供给开发人员。通过使用这些服务，开发人员能够构建出功能丰富、轻便且高效的 BPM 应用程序。

它的特色是提供了 Eclipse 插件，开发人员可以通过插件直接绘画出业务流程图。一个简单的业务流程如图 12-1 所示。

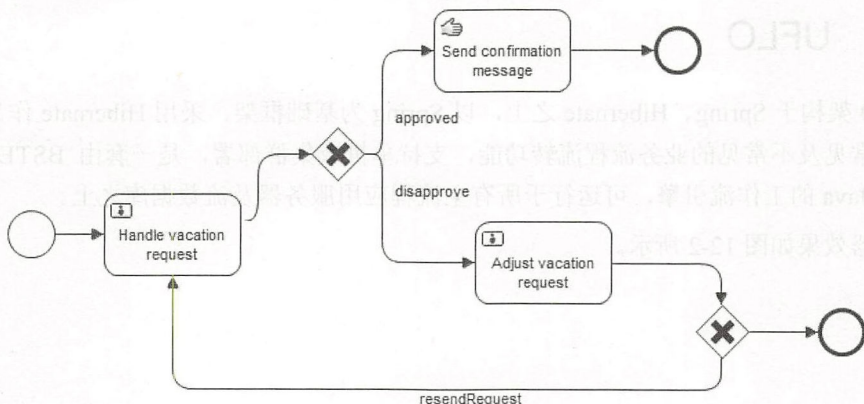


图 12-1 Activiti workflow

关键对象如下。

- **Deployment:** 流程部署对象，部署一个流程时创建。
- **ProcessDefinitions:** 流程定义，部署成功后自动创建。
- **ProcessInstances:** 流程实例，启动流程时创建。
- **Task:** 任务，在 Activiti 中的 Task 仅指有角色参与的任务，即定义中的 UserTask。
- **Execution:** 执行计划，流程实例和流程执行中的所有节点都是 Execution，如 UserTask、ServiceTask 等。

安装流程设计器

在有网络的情况下，安装流程设计器步骤如下：

(1) 打开 “Help” → “Install New Software”。

(2) 在 Install 界面板中，单击 Add 按钮。

(3) 填入以下内容：

Name: Activiti BPMN 2.0 designer

Location: <http://activiti.org/designer/update/>

(4) 回到安装界面，在面板正中列表中把所有展示出来的项目都勾上。

(5) 安装完以后，单击新建工程 “new” → “Other...” 打开面板，就能够看到 Activiti 的流程设计界面。

Activiti 提供相应的设计器，可以匹配自身公司的业务场景，设计自定义的流程，然后将设计完成的 XML 文件上传到工作流引擎中，供相应的类调用。

12.1.2 UFLO

UFLO 架构于 Spring、Hibernate 之上，以 Spring 为基础框架，采用 Hibernate 作为持久层，提供各种常见及不常见的业务流程流转功能，支持单机或集群部署，是一套由 BSTEK 自主研发的基于 Java 的工作流引擎，可运行于所有主流应用服务器及流数据库之上。

设计器效果如图 12-2 所示。

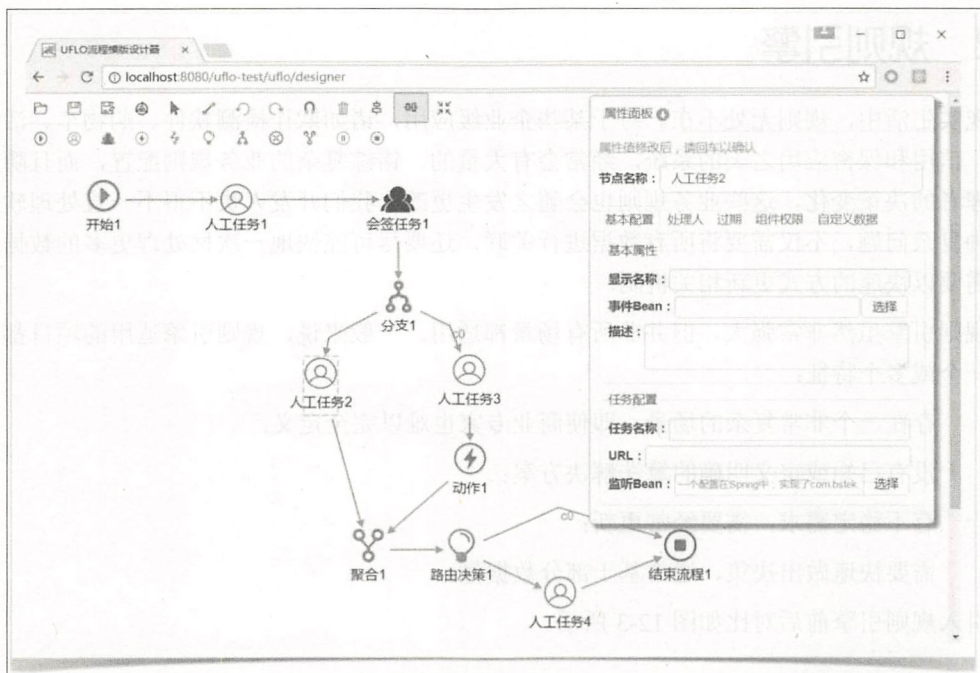


图 12-2 UFLO 流程设计页面

将 UFLO 添加到项目中分两种情况，一种是 Maven 项目，另一种是普通的 Web 项目。如果使用 Maven 的方式，则需要在 pom.xml 文件中加入：

```
<dependency>
  <groupId>com.bstek.uflo</groupId>
  <artifactId>uflo-console</artifactId>
  <version>[version]</version>
</dependency>
```

UFLO 的节点包括开始、结束、人工任务、子流程、路由决策、动作、分支、动态分支及聚合。这些节点能够满足我们在业务中遇到的各种复杂场景。

同时，UFLO2 中提供了集群部署的方式，在大型的应用中提高了并发处理的能力。

对于 workflow 引擎的选择，这里给出了两个目前比较流行的方案，一个是老牌的国外的开源项目，一个是国内的开源项目，对于使用者来说，需要根据自身业务的特点，酌情进行选择。

12.2 规则引擎

现实生活中，规则无处不在。对于某些企业级应用，诸如欺诈检测软件、购物车、活动监视器、信用和保密应用之类的系统，经常会有大量的、错综复杂的业务规则配置，而且随着企业管理者的决策变化，这些业务规则也会随之发生更改。我们开发人员不得不一直处理软件中的各种复杂问题，不仅需要将所有数据进行关联，还要尽可能快地一次性处理更多的数据，甚至还需要以快速的方式更新相关机制。

规则引擎虽然非常强大，但并非所有场景都适用。一般来说，规则引擎适用的项目都具有以下一个或多个特征：

- 存在一个非常复杂的场景，即使商业专家也难以完全定义；
- 没有已知或定义明确的算法解决方案；
- 有不稳定需求，需要经常更新；
- 需要快速做出决策，通常基于部分数据量。

引入规则引擎前后对比如图 12-3 所示。

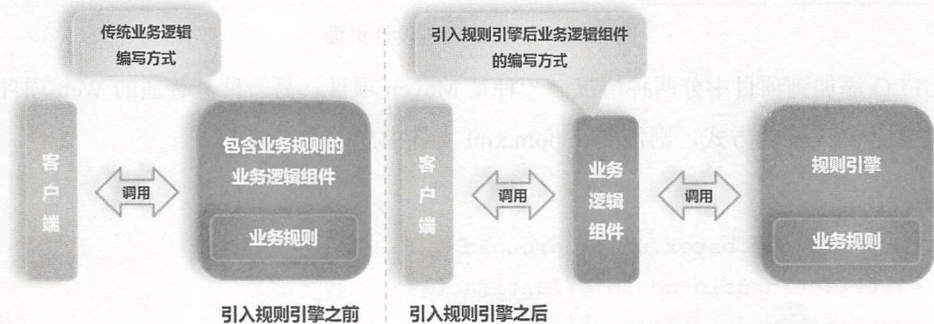


图 12-3 引入规则引擎前后对比

12.2.1 Drools

Drools 是一个基于 Java 的开源规则引擎，可以将复杂多变的规则从硬编码中解放出来，以规则脚本的形式存放在文件中，使得规则的变更不需要修正代码，重启机器就可以立即在线上环境生效。

Drools 规则是在 Java 应用程序上运行的，其要执行的步骤顺序由代码确定。为了实现这一点，Drools 规则引擎将业务规则转换成执行树，每个规则条件分为小块，在树结构中连接和重

用。每次将数据添加到规则引擎中时，它将在与此类似的树中进行求值，并到达一个动作节点，在该节点处，它们将被标记为准备执行特定规则的数据。

Drools 规则引擎基于 ReteOO 算法（对面向对象系统的 Rete 算法进行了增强和优化的实现），它将事实（Fact）与规则进行匹配，以推断相应的规则结果，这个过程称之为模式匹配，如图 12-4 所示。

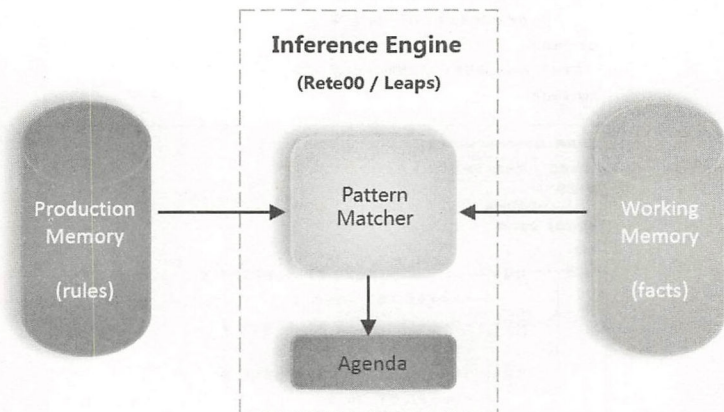


图 12-4 Drools 规则引擎

规则引擎默认不会在规则评估时立即执行业务规则，除非我们强制指定。当到达一个事实（Fact）与规则相匹配的节点时，规则评估会将规则操作与触发数据添加到一个叫作议程（Agenda）的组件中，如果同一个事实（Fact）与多个规则相匹配，则认为这些规则是冲突的，议程（Agenda）使用冲突解决策略（Conflict Resolution strategy）管理这些冲突规则的执行顺序。整个生命周期中，规则评估与规则执行之间有着明确的分割。规则操作的执行可能会导致事实（Fact）的更新，从而与其他规则相匹配，导致它们的触发，称之为前向链接。

12.2.2 URule

URule 是一款基于 RETE 算法的纯 Java 的开源规则引擎产品，提供了向导式规则集、脚本式规则集、决策表、决策树、评分卡及决策流共六种类型的规则定义方式，配合基于 Web 的设计器，可快速实现规则的定义、维护与发布。架构于 Spring 之上，基于浏览器的可视化规则设计器和仿真测试机制。具有完善的版本控制机制和对自然语言的支持，可编写纯中文脚本式规则。

向导式决策集如图 12-5 所示。

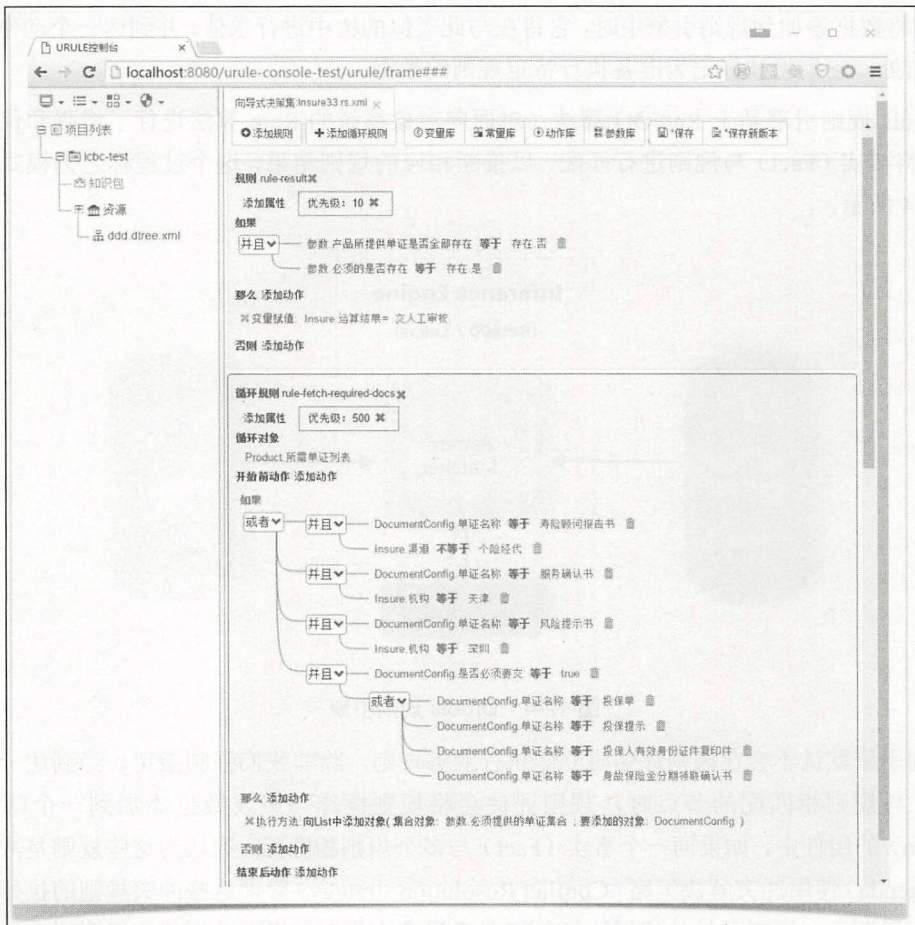


图 12-5 URULE 向导式决策集

我们可以基于 URULE 将业务和规则分开。这一点在项目不断变得庞大的过程中，体现的优势会越来越明显。

对于微服务的应用，规则引擎的选择是非常重要的，因为当业务规模膨胀到一定程度时，业务代码中耦合太多的规则本身就是一件风险非常大的事情。即使有灰度发布等手段，但是过于频繁的发布版本对变化的响应也会非常差。所以，将频繁变化的规则部分抽象出来并基于规则发布，能够更快速地响应需求的变化。

12.3 调度系统

对于微服务而言，一款支持并发且能够横向扩展的调度平台非常重要。

XXL-JOB 是一个轻量级分布式任务调度框架，其核心设计目标是开发迅速、学习简单、轻量级、易扩展。现已开放源代码并接入多家公司线上产品线，开箱即用。

它的特点如下。

- 简单：支持通过 Web 页面对任务进行 CRUD 操作，操作简单，一分钟上手。
- 动态：支持动态修改任务状态、暂停/恢复任务，以及终止运行中任务，即时生效。
- 调度中心 HA（中心式）：调度采用中心式设计，“调度中心”基于集群 Quartz 实现并支持集群部署，可保证调度中心 HA。
- 执行器 HA（分布式）：任务分布式执行，任务“执行器”支持集群部署，可保证任务执行 HA。
- 注册中心：执行器会周期性自动注册任务，调度中心将会自动发现注册的任务并触发执行。同时，也支持手动录入执行器地址。
- 弹性扩容缩容：一旦有新执行器机器上线或者下线，下次调度时将会重新分配任务。
- 路由策略：执行器集群部署时提供丰富的路由策略，包括第一个、最后一个、轮询、随机、一致性 HASH、最不经常使用、最近最久未使用、故障转移、忙碌转移等。
- 故障转移：任务路由策略选择“故障转移”情况下，如果执行器集群中某一台机器故障，则会自动“Failover”切换到一台正常的执行器发送调度请求。
- 失败处理策略：调度失败时的处理策略，策略包括失败告警（默认）、失败重试。
- 失败重试：调度中心调度失败且启用“失败重试”策略时，会自动重试一次；执行器执行失败且回调失败重试状态时，也会自动重试一次。
- 阻塞处理策略：调度过于密集执行器来不及处理时的处理策略，策略包括单机串行（默认）、丢弃后续调度、覆盖之前调度。
- 分片广播任务：执行器集群部署时，任务路由策略选择“分片广播”情况下，一次任务调度将会广播触发集群中所有执行器执行一次任务，可根据分片参数开发分片任务。
- 动态分片：分片广播任务以执行器为维度进行分片，支持动态扩容执行器集群从而动态增加分片数量，协同进行业务处理；在进行大数据量业务操作时可显著提升任务处理能力和速度。
- 事件触发：除了“Cron 方式”和“任务依赖方式”触发任务执行，还支持基于事件的触发任务方式。调度中心提供触发任务单次执行的 API 服务，可根据业务事件灵活触发。

- 任务进度监控：支持实时监控任务进度。
- Rolling 实时日志：支持在线查看调度结果，并且支持以 Rolling 方式实时查看执行器输出的完整的执行日志。
- GLUE：提供 Web IDE，支持在线开发任务逻辑代码，动态发布，实时编译生效，省略部署上线的过程，支持 30 个版本的历史版本回溯。
- 脚本任务：支持以 GLUE 模式开发和运行脚本任务，包括 Shell、Python、Node.js 等类型脚本。
- 任务依赖：支持配置子任务依赖，当父任务执行结束且执行成功后会主动触发一次子任务的执行，多个子任务用逗号分隔。
- 一致性：“调度中心”通过 DB 锁保证集群分布式调度的一致性，一次任务调度只会触发一次执行。
- 自定义任务参数：支持在线配置调度任务入参，即时生效。
- 调度线程池：调度系统多线程触发调度运行，确保调度精确执行，不被堵塞。
- 数据加密：调度中心和执行器之间的通信进行数据加密，提升调度信息安全性。
- 邮件报警：任务失败时支持邮件报警，支持配置多邮件地址群发报警邮件。
- 推送 Maven 中央仓库：将会把最新稳定版推送到 Maven 中央仓库，方便用户接入和使用。
- 运行报表：支持实时查看运行数据，如任务数量、调度次数、执行器数量等，以及调度报表，如调度日期分布图、调度成功分布图等。

安装过程也非常简单，下载项目源码并解压，获取“调度数据库初始化 SQL 脚本”并执行即可。

项目的开源地址：<https://github.com/xuxueli/xxl-job>。

启动成功后的运行效果如图 12-6 所示。

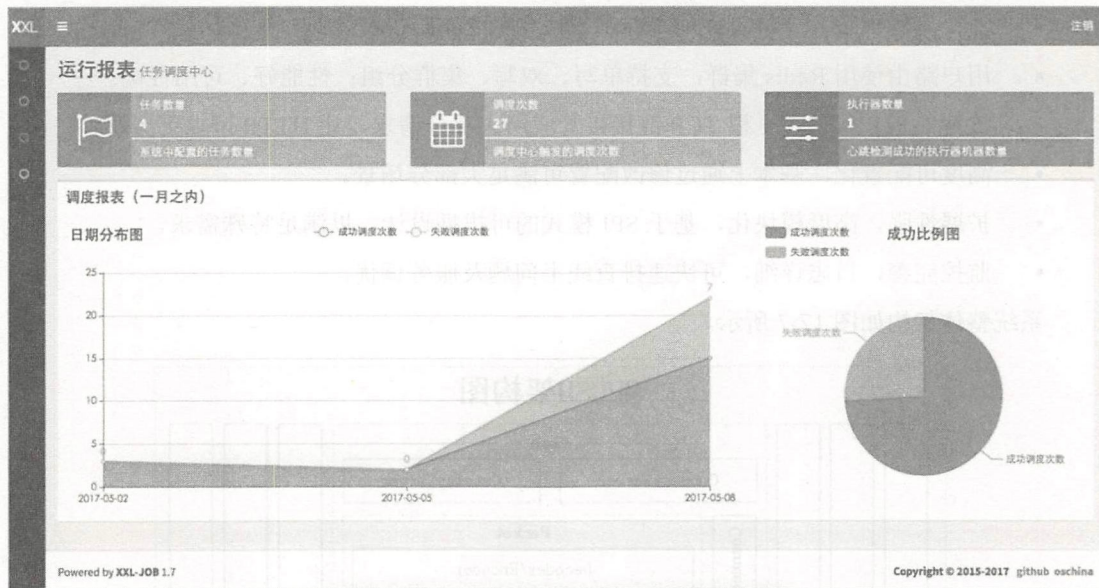


图 12-6 XXL-JOB 运行界面

分布式系统中，构建系统间的服务处理需要成百上千的调度，能够有效地控制调度的规模，保证调度的可用性非常重要。

12.4 消息推送

微服务中，除了正常的通信接口，一款支持多平台的消息推送平台也非常重要。

MPUSH 是一款开源的实时消息推送系统，采用 Java 语言开发，服务端采用模块化设计，具有协议简洁、传输安全、接口流畅、实时高效、扩展性强、可配置化、部署方便、监控完善等特点，同时也是少有的可商用的开源推送系统。

开源地址：<https://github.com/mpusher/>。

它具有如下特性和优势：

- 源码全部开放，包括 Server、Android、iOS。
- 代码质量高，全部模块化设计，真正的商用级产品，考虑到了推送中遇到的大部分场景。
- 安全性高，基于 RSA 精简的加密握手协议，简单、高效、安全。
- 支持断线重连及弱网下的快速重连，无网络下自动休眠以节省电量和资源。
- 协议简洁，接口流畅，支持数据压缩，更加节省流量。

- 支持集群部署，支持负载均衡，基于成熟的 ZooKeeper 实现。
- 用户路由使用 Redis 集群，支持单写、双写、集群分组；性能好、可用性高。
- 支持 HTTP 代理，一根 TCP 链接接管应用大部分请求，让 HTTP 请求更加及时。
- 高度可配置化，基本上通过修改配置可满足大部分场景。
- 扩展性强，高度模块化，基于 SPI 模式的可拔插设计，以满足特殊需求。
- 监控完善，日志详细，可快速排查线上问题及服务调优。

系统整体架构如图 12-7 所示。

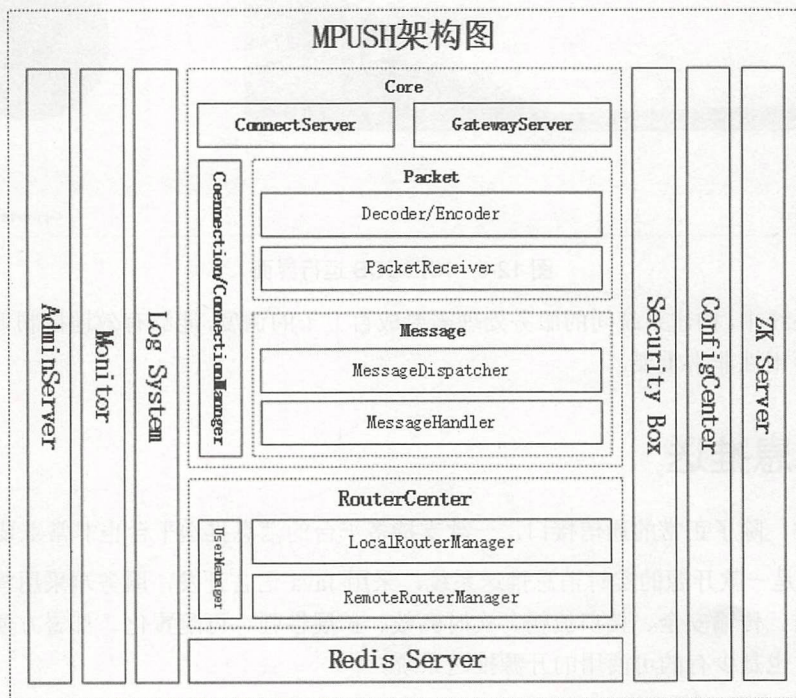


图 12-7 MPUSH 架构

(1) 最左侧三大组件分别是日志系统、监控系统和控制台治理服务。

- **Log System** 主要负责业务日志的输出，主要有连接相关日志、推送链路日志、心跳日志、监控日志等。
- **Monitor** 主要用于系统状态监控，可用于系统调优，包括 JVM 内存、线程、线程池、系统堆栈、垃圾回收、内存泄漏等。
- **AdminServer** 主要用于在控制台对单台机器进行控制与探查，比如查看连接数量、在线用户数，取消本级 ZK 注册，关闭服务等。

(2) 右侧三个分别是 ZK 服务、配置中心和安全工具箱。

- ZK Client 主要负责注册长连接 ip:port、网关 ip:port，以及监听各个节点变化，同时增加了缓存。
- ConfigCenter 是 MPUSH Server 配置化的关键，贯穿于各个模块，非常重要。
- Sercuity Box 主要实现了 RSA 加密、DES 加密、会话密钥生成及 Session 复用（用于快速重连）。

(3) Core 模块分别是长连接服务、网关服务、Packet 编解码、分发模块、Message 序列化及处理模块。

- ConnectServer 用于维持和客户端之间的 TCP 通道，主要负责和客户端交互。
- GatewayServer 用于处理 Mpush Server 之间的消息交互，比如踢人、发送 PUSH。
- Packet 主要是协议部分的编解码和包的完整性校验、最大长度校验等。
- PacketReceiver 主要负责消息的分发，分发是根据 Command 来的。
- Connection/ConnectionManager 主要负责连接管理，定时检查链接空闲情况，是否读写超时，如果连接断开则发出相应的事件给路由中心去处理。
- Message 部分是整个的业务核心，处理消息的序列化，还有压缩、加密等，MessageHandler 会根据不同消息独立处理自己所属的业务，主要有心跳响应、握手及密钥交换、快速重连、绑定/解绑用户、HTTP 代理、消息推送等。

(4) 路由中心主要包括本地路由、远程路由和用户在线管理三大块。

- LocalRouterManager 负责维护用户+设备与连接（connection）之间的关系。
- RemoteRouterManager 负责维护用户+设备与连接所在机器 IP 之间的关系。
- UserManager 主要处理用户上下线事件的广播，以及单台机器的在线用户及数量的维护和查询。

(5) MPUSH 的缓存部分，目前只支持 Redis、支持双写、主备、Hash 等特性。

业务系统是要发送业务消息的服务，所有要推送的消息直接转给 MPNS。

- MPNS 是业务推送系统，负责消息推送、长连接的检查、离线消息存储、用户打标等。
- APNS、JPUSH、MPUSH 等分别是客户端已经接入的推送系统。
- MPNS 主要是为了隔离业务系统和各种推送系统，用户使用哪个长连接服务，业务系统不需要感知，统一由 MPNS 去选择、切换。
- Alloc 负责调度维护 MPushServer 集群，提供查询可用机器列表的接口。

使用 MPUSH 可以方便地构建自己的消息推送平台。

12.5 网关中间件

常用的网关中间件有 Zuul、Kong、Tyk 和 Orange 等。

12.5.1 Orange

Orange 是一个基于 OpenResty 的 API Gateway，提供 API 及自定义规则的监控和管理，如访问统计、流量切分、API 重定向、API 鉴权、Web 防火墙等功能。Orange 可用来替代前置机中广泛使用的 Nginx/OpenResty，在应用服务上前置一个功能丰富的网关系统。它有以下特性：

- 动态更新 Nginx/OpenResty 配置而无须重启或 reload；
- 通过 MySQL 存储来简单支持集群部署；
- 支持多种条件匹配和变量提取；
- 支持通过自定义插件方式扩展功能；
- 内置多个通用插件；
- 全局状态统计；
- 自定义监控；
- URL 重写；
- URI 重定向；
- 访问限速；
- Key based rate limiting；
- HTTP Basic Auth；
- HTTP Key Auth；
- Signature Auth；
- 简单防火墙 WAF；
- 代理、ABTesting、分流；
- Shared Dict 存取接口；
- 提供 Dashboard 用于管理内置插件；
- 开放 API: 所有插件均开放 API 供第三方使用, 通过这些 API 可简单灵活地配置插件、查看运行状态、统计数据等；
- 架构简单，依赖少，许可协议宽松，适合直接二次开发或改造。

安装完成的网关界面如图 12-8 所示。

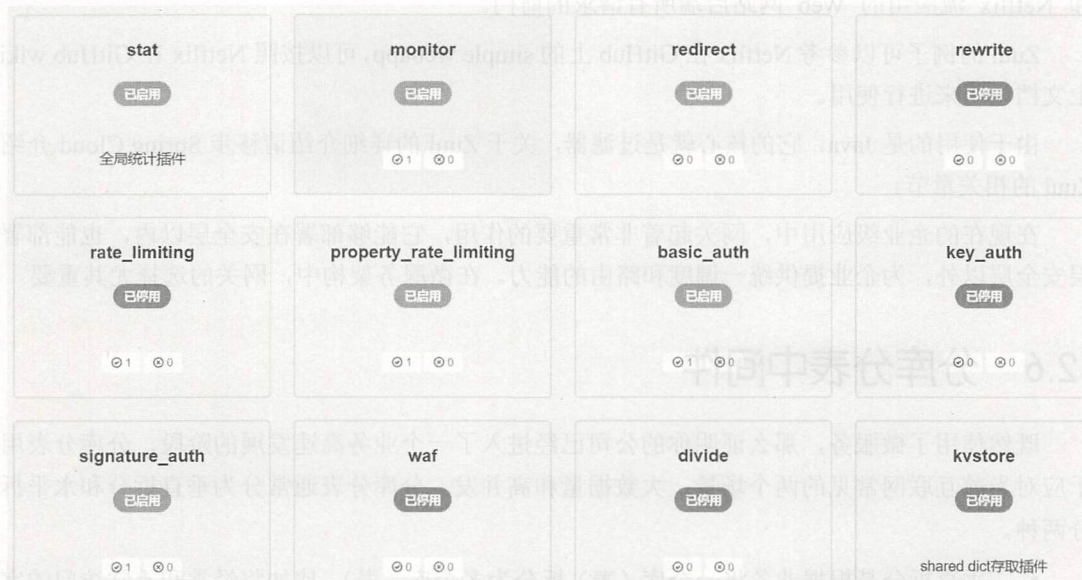


图 12-8 Orange 网关界面

可以根据自身项目需要使用相关的插件。

使用文档地址：<http://orange.sumory.com/docs/>。

12.5.2 Kong

Kong 提供了 API 管理功能，以及围绕 API 管理实现了一些默认的插件，另外还具备集群水平扩展能力，从而提升整体吞吐量。Kong 本身是基于 OpenResty 的，可以在现有 Kong 的基础上进行一些扩展，从而实现更复杂的特性。

Kong 是由 Mashape 公司开源的，基于 Nginx 的 API Gateway。

它的特点是可扩展，支持分布式、模块化。

可以使用 kong-dashboard 对 Kong 的插件和操作进行管理，也可以根据自己公司的业务开发相应的插件。

12.5.3 Zuul

Zuul 是 Netflix 开源的一个 API Gateway 服务器，本质上是一个 Web Servlet 应用。

Zuul 是在云平台上提供动态路由、监控、弹性、安全等边缘服务的框架。Zuul 相当于设备和 Netflix 流应用的 Web 网站后端所有请求的前门。

Zuul 的例子可以参考 Netflix 在 GitHub 上的 simple webapp, 可以按照 Netflix 在 GitHub wiki 上文档说明来进行使用。

由于使用的是 Java, 它的核心就是过滤器, 关于 Zuul 的详细介绍请移步 Spring Cloud 介绍 Zuul 的相关章节。

在现在的企业级应用中, 网关起着非常重要的作用, 它能够部署在安全层以内, 也能部署层安全层以外, 为企业提供统一调度和路由的能力。在微服务架构中, 网关的选择尤其重要。

12.6 分库分表中间件

既然使用了微服务, 那么证明你的公司已经进入了一个业务高速发展的阶段, 分库分表用于应对当前互联网常见的两个场景: 大数据量和高并发。分库分表通常分为垂直拆分和水平拆分两种。

- 垂直拆分是根据业务将一个库(表)拆分为多个库(表), 比如将经常和不常访问的字段拆分至不同的库或表中。由于与业务关系密切, 目前的分库分表产品均使用水平拆分方式。
- 水平拆分则是根据分片算法将一个库(表)拆分为多个库(表), 比如按照 ID 的最后一位以 3 取余, 尾数是 1 的放入第 1 个库(表), 尾数是 2 的放入第 2 个库(表)。

12.6.1 Sharding-JDBC

Sharding-JDBC 是当当应用框架 ddframe 中, 从关系型数据库模块 dd-rdb 中分离出来的数据库水平分片框架, 实现透明化数据库分库分表访问。Sharding-JDBC 是继 Dubbox 和 elastic-job 之后, ddframe 系列开源的第 3 个项目。

Sharding-JDBC 直接封装 JDBC API, 可以理解为增强版的 JDBC 驱动, 旧代码迁移成本几乎为零, 它的几个适用场景如下:

- 适用于任何基于 Java 的 ORM 框架, 如 JPA、Hibernate、Mybatis、Spring JDBC Template 或直接使用 JDBC。
- 适用于任何第三方的数据库连接池, 如 DBCP、C3P0、BoneCP、Druid 等。
- 适用于任意实现 JDBC 规范的数据库。虽然目前仅支持 MySQL, 但已有支持 Oracle、SQLServer 等数据库的计划。

Sharding-JDBC 定位为轻量 Java 框架,使用客户端直连数据库,以 jar 包形式提供服务,无 Proxy 代理层,无须额外部署,无其他依赖,DBA 也无须改变原有的运维方式。

Sharding-JDBC 分片策略灵活,可支持等号、between、in 等多维度分片,也可支持多分片键。

SQL 解析功能完善,支持聚合、分组、排序、limit、or 等查询,并支持 Binding Table 和笛卡儿积表查询。

➤ 分片规则配置

Sharding-JDBC 的分片逻辑非常灵活,支持分片策略自定义、复数分片键、多运算符分片等功能。

比如,根据用户 ID 分库,根据订单 ID 分表这种分库分表结合的分片策略;或根据年分库,月份+用户区域 ID 分表这样的多片键分片。

Sharding-JDBC 除了支持等号运算符进行分片,还支持 in/between 运算符分片,提供了更加强大的分片功能。

Sharding-JDBC 提供了 Spring 命名空间用于简化配置,以及规则引擎用于简化策略编写。由于目前刚开源分片核心逻辑,这两个模块暂未开源,待核心稳定后将开源其他模块。

➤ JDBC 规范重写

Sharding-JDBC 对 JDBC 规范的重写思路是针对 DataSource、Connection、Statement、PreparedStatement 和 ResultSet 五个核心接口进行封装,将多个真实 JDBC 实现类集合(如 MySQL JDBC 实现/DBCP JDBC 实现等)纳入 Sharding-JDBC 实现类管理。

Sharding-JDBC 尽量最大化实现 JDBC 协议,包括 addBatch 这种在 JPA 中会使用的批量更新功能。但分片 JDBC 毕竟与原生 JDBC 不同,所以目前仍有未实现的接口,包括 Connection 游标、存储过程和 savePoint 相关、ResultSet 向前遍历和修改等不太常用的功能。此外,为了保证兼容性,并未实现 JDBC 4.1 及其后发布的接口(如 DBCP 1.x 版本不支持 JDBC 4.1)。

➤ SQL 解析

SQL 解析作为分库分表类产品的核心,性能和兼容性是最重要的衡量指标。目前常见的 SQL 解析器主要有 fdb/jsqlparser 和 Druid。Sharding-JDBC 使用 Druid 作为 SQL 解析器,经实际测试,Druid 解析速度是另外两个解析器的几十倍。

目前 Sharding-JDBC 支持 join、aggregation (包括 avg)、order by、group by、limit、甚至 or 查询等复杂 SQL 的解析。目前不支持 union、部分子查询、函数内分片等不太应在分片场景中出现的 SQL 解析。

➤ SQL 改写

SQL 改写分为两部分,一部分是将分表的逻辑表名称替换为真实表名称,另一部分是根据

SQL 解析结果替换一些在分片环境中不正确的功能。这里举两个例子：

第 1 个例子是 avg 计算。在分片的环境中，以 $\text{avg1} + \text{avg2} + \text{avg3} / 3$ 计算平均值并不正确，需要改写为 $(\text{sum1} + \text{sum2} + \text{sum3}) / (\text{count1} + \text{count2} + \text{count3})$ 。这就需要将包含 avg 的 SQL 改写为 sum 和 count，在将结果归并时重新计算平均值。

第 2 个例子是分页。假设每 10 条数据为一页，取第 2 页数据。在分片环境下获取 “limit 10, 10”，归并之后再根据排序条件取出前 10 条数据是不正确的结果。正确的做法是将分条件改写为 “limit 0, 20”，取出所有前 2 页数据，再结合排序条件算出正确的数据。可以看到越是靠后的 limit，分页效率就会越低，也越浪费内存。有很多方法可避免使用 limit 进行分页，比如构建记录行记录数和行偏移量的二级索引，或使用上次分页数据结尾 ID 作为下次查询条件的分页方式。

➤ SQL 路由

SQL 路由是根据分片规则配置，将 SQL 定位至真正的数据源。主要分为单表路由、Binding 表路由和笛卡儿积路由。

单表路由最简单，但路由结果不一定落入唯一库（表），因为支持根据 between 和 in 这样的操作符进行分片，所以最终结果仍然可能落入多个库（表）。

Binding 表可理解为分库分表规则完全一致的主从表。举例说明：订单表和订单详情表都根据订单 ID 作为分片键，任意时刻分片逻辑均相同。这样的关联查询和单表查询难度和性能相当。

笛卡儿积查询最为复杂，因为无法根据 Binding 关系定位分片规则的一致性，所以非 Binding 表的关联查询需要拆解为笛卡儿积组合执行。查询性能较低，而且数据库连接数较高，需谨慎使用。

➤ SQL 执行

路由至真实数据源后，Sharding-JDBC 将采用多线程并发执行 SQL，并完成对 addBatch 等批量方法的处理。

➤ 结果归并

结果归并包括 4 类：普通遍历类、排序类、聚合类和分组类。每种类型都会先根据分页结果跳过不需要的数据。

普通遍历类最为简单，只需按顺序遍历 ResultSet 的集合即可。

排序类结果将结果先排序再输出，因为各分片结果均按照各自条件完成排序，所以采用归并排序算法整合最终结果。

聚合类分为比较型、累加型和平均值型 3 种类型。比较型包括 max 和 min，只返回最大（小）结果，累加型包括 sum 和 count，需要将结果累加后返回；平均值则通过 SQL 改写的 sum 和 count 计算得出。

分组类最为复杂，需要把所有的 `ResultSet` 结果放入内存，使用 `map-reduce` 算法分组，最后根据排序和聚合条件做相关处理。最消耗内存、最损失性能的部分即是此，可以考虑使用 `limit` 合理地限制分组数据大小。

结果归并部分目前并未采用管道解析的方式，之后会针对这里做更多改进。

12.6.2 MyCat

MyCat 是一个开源的分布式数据库系统，是一个实现了 MySQL 协议的服务器，前端用户可以把它看作一个数据库代理，用 MySQL 客户端工具和命令行访问，而其后端可以用 MySQL 原生协议与多个 MySQL 服务器通信，也可以用 JDBC 协议与大多数主流数据库服务器通信，其核心功能是分表分库，即将一个大表水平分割为 N 个小表，存储在后端 MySQL 服务器或者其他数据库里。

MyCat 是一个强大的数据库中间件，不仅仅可以用作读写分离，以及分表分库、容灾备份，而且可以用于多租户应用开发、云平台基础设施，让你的架构具备很强的适应性和灵活性。借助于即将发布的 MyCat 智能优化模块，系统的数据访问瓶颈和热点一目了然，根据这些统计分析数据，你可以自动或手工调整后端存储，将不同的表映射到不同存储引擎上，而整个应用的代码一行也不用改变。

MyCat 的原理中最重要的一个动词是“拦截”，它拦截了用户发送过来的 SQL 语句，首先对 SQL 语句做了一些特定的分析，如分片分析、路由分析、读写分离分析、缓存分析等，然后将此 SQL 发往后端的真实数据库，并将返回的结果做适当的处理，最终再返回给用户。

当 MyCat 收到一个 SQL 时，会先解析这个 SQL，查找涉及的表，然后看此表的定义，如果有分片规则，则获取 SQL 里分片字段的值，并匹配分片函数，得到该 SQL 对应的分片列表，然后将 SQL 发往这些分片去执行，最后收集和处理所有分片返回的结果数据，并输出到客户端。以 `select * from Orders where prov=?` 语句为例，查到 `prov=wuhan`，按照分片函数，`wuhan` 返回 `dn1`，于是 SQL 就发给了 MySQL1，去取 DB1 上的查询结果，并返回给用户。

MyCat 发展到现在，适用的场景已经很丰富，而且不断有新用户给出创新性的方案，以下是几个典型的应用场景：

- 单纯的读写分离，此时配置最为简单，支持读写分离，主从切换。
- 分表分库，对于超过 1000 万的表进行分片，最大支持 1000 亿的单表分片。
- 多租户应用，每个应用一个库，但应用程序只连接 MyCat，从而不改造程序本身，实现多租户化。

- 报表系统，借助于 MyCat 的分表能力，处理大规模报表的统计。
- 代替 Hbase，分析大数据。
- 作为海量数据实时查询的一种简单有效方案，比如 100 亿条频繁查询的记录需要在 3 秒内查询出来结果，除了基于主键的查询，还可能存在范围查询或其他属性查询，此时 MyCat 可能是最简单有效的选择。

数据的切分（Sharding）根据其切分规则的类型，可以分为两种切分模式。一种是按照不同的表（或者 Schema）来切分到不同的数据库（主机）上，这种切可以称之为数据的垂直（纵向）切分；另外一种则是根据表中的数据的关系，将同一个表中的数据按照某种条件拆分到多台数据库（主机）上，这种切分称之为数据的水平（横向）切分。

垂直切分的最大特点就是规则简单，实施也更为方便，尤其适合各业务之间的耦合度非常低，相互影响很小，业务逻辑非常清晰的系统。在这种系统中，可以很容易做到将不同业务模块所使用的表拆分到不同的数据库中。根据不同的表来进行拆分，对应用程序的影响也更小，拆分规则也会比较简单清晰。

水平切分与垂直切分相比，相对来说稍微复杂一些。因为要将同一个表中的不同数据拆分到不同的数据库中，对于应用程序来说，拆分规则本身就比较根据表名来拆分更为复杂，后期的数据维护也会更为复杂一些。

12.7 报表引擎

UReport2 是一款基于架构在 Spring 之上的纯 Java 的高性能报表引擎，通过迭代单元格可以实现任意复杂的中国式报表。相比 UReport1，UReport2 重写了全部代码，弥补了 UReport1 在功能及性能上的各种不足。

在 UReport2 中，提供了全新的基于网页的报表设计器，可以在 Chrome、Firefox、Edge 等各种主流浏览器运行（IE 浏览器除外）。使用 UReport2，打开浏览器即可完成各种复杂报表的设计制作。

UReport2 的设计器是基于网页的，所以我们配置好一个项目，也就完成了报表设计器的安装，因为 UReport2 是一款纯 Java 的报表引擎，所以它支持现在流行的所有类型 J2EE 项目，这里主要介绍基于 Maven 的 J2EE 项目中如何包含 UReport2。

配置完成后，UReport2 的设计界面如图 12-9 所示。

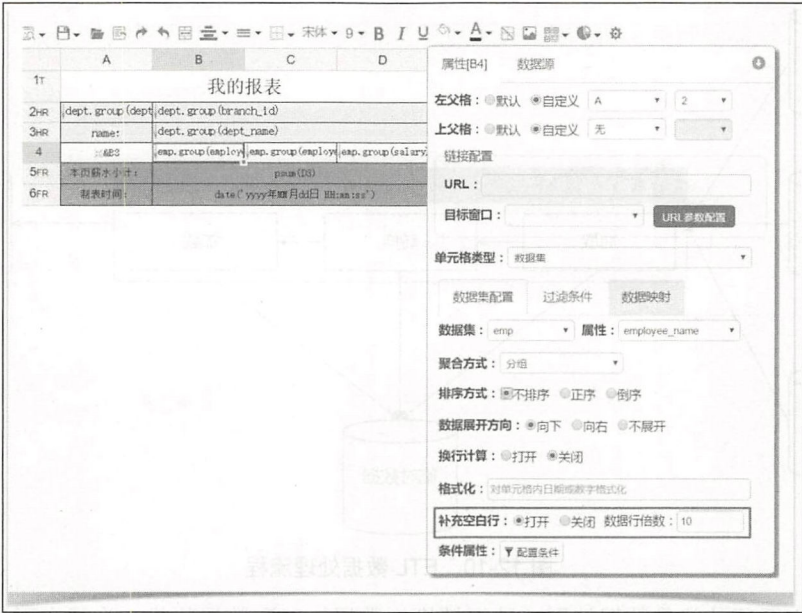


图 12-9 UReport2 报表设计界面

通过对类似 Excel 表格的设置，我们就可以生成动态的页面，这样会更加高效地应对复杂的业务场景和需求的不断变化。

12.8 数据处理

ETL 中三个字母分别代表的是 Extract、Transform、Load，即抽取、转换、加载。

- (1) 数据抽取：从源数据源系统抽取目的数据源系统需要的数据。
- (2) 数据转换：将从源数据源获取的数据按照业务需求，转换成目的数据源要求的形式，并对错误、不一致的数据进行清洗和加工。
- (3) 数据加载：将转换后的数据装载到目的数据源。

ETL 原本是作为构建数据仓库的一个环节，负责将分布的、异构数据源中的数据（如关系数据、平面数据文件等）抽取到临时中间层后进行清洗、转换、集成，最后加载到数据仓库或数据集中，成为联机分析处理、数据挖掘的基础。现在也越来越多地将 ETL 应用于一般信息系统中数据的迁移、交换和同步。一个简单的 ETL 体系如图 12-10 所示。

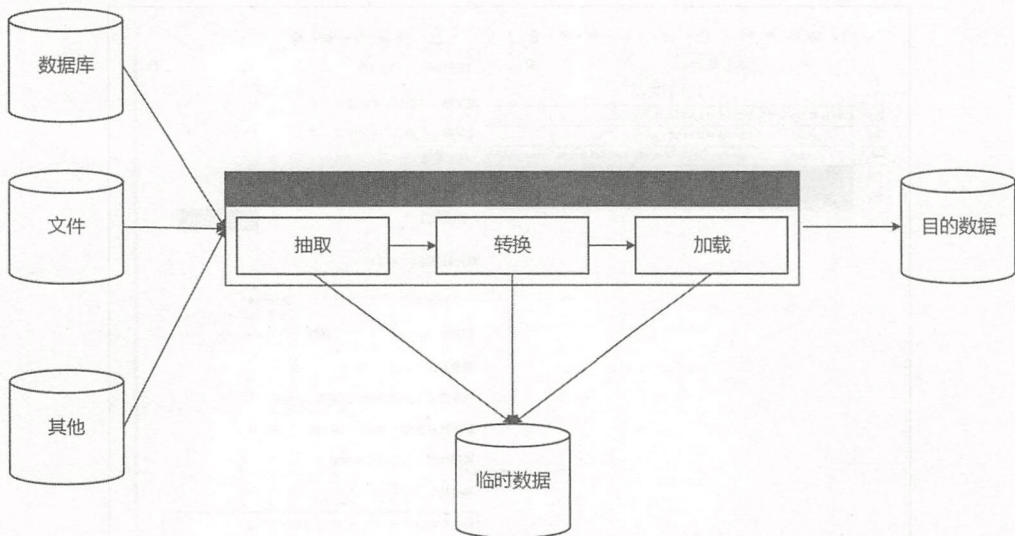


图 12-10 ETL 数据处理流程

ETL 的主要环节就是数据抽取、数据转换、数据加工和数据装载。为了实现这些功能，各个 ETL 工具一般会进行一些功能上的扩充，如 workflow、调度引擎、规则引擎、脚本支持、统计信息等。

12.8.1 Spring Batch

Spring Batch 是一个轻量级的完全面向 Spring 的批处理框架，可以应用于企业级大规模的数据处理系统。Spring Batch 以 POJO 和开发者熟知的 Spring 框架为基础，使开发者更容易地访问和利用企业级服务。Spring Batch 可以提供大量的、可重复的数据处理功能，包括日志记录/跟踪、事务管理、作业处理统计、任务重新启动/跳过，以及资源管理等重要功能。

业务方案如下。

- 批处理定期提交。
- 并行批处理：并行处理工作。
- 企业消息驱动处理。
- 大规模的并行处理。
- 手动或是有计划的重启。
- 局部处理：跳过记录（如回滚）。

技术目标如下。

- 利用 Spring 编程模型，使程序员专注于业务处理，让 Spring 框架管理流程。
- 明确分离批处理的执行环境和应用。
- 提供核心的、共通的接口。
- 提供开箱即用（out of the box）的、简单的、默认的核心执行接口。
- 提供 Spring 框架中配置、自定义、和扩展服务。
- 所有存在的核心服务可以很容易地被替换和扩展，不影响基础层。
- 提供一个简单的部署模式，利用 Maven 构建独立的 jar 文件。

这种分层结构有三个重要的组成部分：应用层、核心层、基础架构层。应用层包含所有的批处理作业，通过 Spring 框架管理程序员自定义的代码。核心层包含了 Batch 启动和控制所需要的核心类，如 JobLauncher、Job 和 Step 等。应用层和核心层建立在基础架构层之上，基础架构层提供共通的读（ItemReader）、写（ItemWriter）和服务（如 RetryTemplate：重试模块，可以被应用层和核心层使用）。

Spring batch 的执行过程如图 12-11 所示。

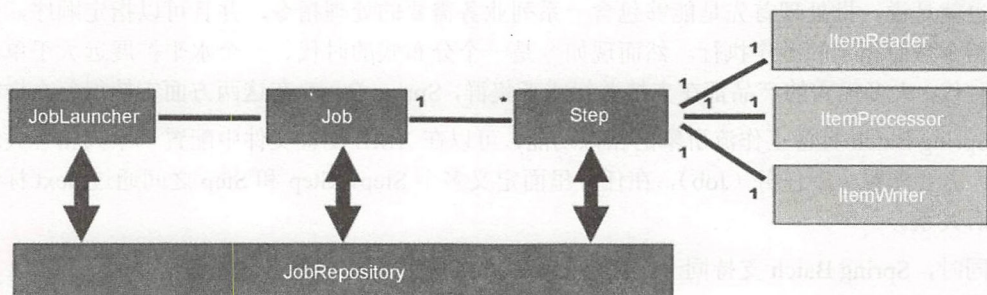


图 12-11 Spring Batch 的执行流程

每个 Batch 都会包含一个 Job。Job 就像一个容器，这个容器里装了若干 Step，Batch 中实际干活的也就是这些 Step，至于 Step 干什么活，无外乎读取数据、处理数据，然后将这些数据存储起来（ItemReader 用来读取数据，ItemProcessor 用来处理数据，ItemWriter 用来写数据）。JobLauncher 用来启动 Job，JobRepository 是上述处理提供的一种持久化机制，它为 JobLauncher、Job 和 Step 实例提供 CRUD 操作。

外部控制器调用 JobLauncher 启动一个 Job，Job 调用自己的 Step 去实现对数据的操作，Step 处理完成后，再将处理结果一步步返回给上一层，这就是 Batch 处理实现的一个简单流程。

批处理的典型概念模型的设计非常精简，下面的 10 个关键词完整支撑了整个框架。Spring Batch 概念及介绍如表 12-1 所示。

表 12-1 Spring Batch 概念及介绍

关 键 词	描 述
Job repository	基础组件，用来持久化 Job 数据，默认使用内存
Job launcher	基础组件，用来启动 Job
Job	应用组件，是 Spring Batch 操作的基础单元
Step	Job 的一个阶段，Job 由一组 Step 组成
Tasklet	Step 的一个事务过程，包括重复执行、同步、异步等策略
Item	从数据源读出或写入的一条数据记录
Chunk	给定数量的 Item 集合
Item Reader	从给定的数据源读取 Item 集合
Item Processor	在 Item 写入数据源之前进行数据清洗、转换、校验、过滤等
Item Writer	把 Chunk 中包含的 Item 写入数据源

Spring Batch 用来做批处理，批处理最早基本用来指脚本，比如批处理脚本，就是把一系列的操作写到一个 bat 或者 sh 文件里面，一次调用，脚本里面的一系列操作就按照顺序逐条指令执行了。

也就是说，批处理首先是能够包含一系列业务需要的处理指令，并且可以指定顺序，然后这些指令按照指定的顺序执行。然而现如今是一个分布式的时代、一个水平扩展远大于单机性能的时代，大多优秀的产品都在支持并行或者集群，Spring Batch 在这两方面支持得怎么样呢？

Spring Batch 具备工作流引擎的相似功能，可以在 XML 配置文件中配置一系列相互关联的步骤。需要配置一个任务（Job），在任务里面定义多个 Step，Step 和 Step 之间通过 next 标签指明前后关系。

同时，Spring Batch 支持同一个任务的多线程、多进程、多服务器处理。

12.8.2 Kettle

Kettle 是一个 ETL（Extract、Transform and Load，抽取、转换和加载）工具，ETL 工具在数据仓库项目中使用得很频繁，Kettle 也能够应用在下面一些场景中：

- 在不同应用或数据库之间整合数据；
- 把数据库中的数据导出到文本文件；
- 大批量数据装加载数据库；
- 数据清洗；
- 集成应用相关项目是个使用。

Kettle 是一款国外开源的 ETL 工具，纯 Java 编写，可以在 Windows、Linux、UNIX 上运行，

无须安装，数据抽取高效稳定。中文名称叫水壶，该项目的主程序员 MATT 希望把各种数据放到一个壶里，然后以一种指定的格式流出。它是一个 ETL 工具集，允许管理来自不同数据库的数据，通过提供一个图形化的用户环境来描述你想做什么，而不是你想怎么做。

Kettle 中有 transformation 和 job 两种脚本文件，transformation 完成针对数据的基础转换，job 则完成整个工作流的控制。

Kettle 有如下几个概念。

作业：

一个 transformation（转换）就是一个 ETL 的过程，而 job（作业）则是多个 transformation（转换）、job（作业）的集合，在 job（作业）中可以对 transformation（转换）或 job（作业）进行调度、定时任务等。

转换：

转换通过图形化的页面，方便直观地让你完成数据转换的操作。

设计好一个转换后，转换保存的本地文件是.ktr 文件。从.ktr 文件中可以读取出该转换的元数据 transMeta。

12.9 并发编程

在微服务中，性能问题是一个非常大的问题，随着业务量的增长，查询和操作接口都会面临非常大的挑战。

Akka 是 JVM 平台上构建高并发、分布式和容错应用的工具包和运行时。Akka 用 Scala 语言写成，同时提供了 Scala 和 Java 的开发接口。

使用 Akka 框架编写的应用程序既可以横向扩展（Scale Out），也可纵向扩展（Scale Up）。

Akka 框架意在简化高并发、可扩展及分布式应用程序的设计，它具有如下优势：

- 编写并发应用程序更简单，Akka 提供了更高的抽象，开发人员只需要专注于业务逻辑，而无须像 Java 语言那样需要处理低级语义如线程、锁及非阻塞 I/O 等。
- 高容错，Akka 使用“let it crashes”机制，当 Actor 出错时可以快速恢复。
- 事件驱动的架构，Akka 中的 Actor 之间的通信采用异步消息发送，能够完美支持事件驱动。
- 位置透明，无论 Actor 运行在本地机器还是远程机器上，对于用户来说都是透明的，这极大地简化了多核处理器和分布式系统上的应用程序编程。
- 事务支持能力，支持软件事务内存（software transactional memory, STM），使 Actor

具有原子消息流的操作能力。

Akka 框架由下列 10 个组件构成。

- akka-actor: 包括经典的 Actor、Typed Actors、I/O Actor 等。
- akka-remote: 远程 Actor。
- akka-testkit: 测试 Actor 系统的工具箱。
- akka-kernel: Akka 微内核, 用于运行精简的微型应用程序服务器, 无须运行于 Java 应用服务器上。
- akka-transactor: Transactors 即支持事务的 Actors, 集成了 Scala STM。
- akka-agent: 代理, 同样集成了 Scala STM。
- akka-camel: 集成 Apache Camel。
- akka-zeromq: 集成 ZeroMQ 消息队列。
- akka-slf4j: 支持 SLF4J 日志功能。
- akka-filebased-mailbox: 支持基于文件的 mailbox。

12.10 分布式配置

在 Spring Cloud 中, 由 Spring Cloud config 实现分布式配置, 其实有已经集成好的分布式配置的项目可供我们选择。

12.10.1 Disconf

Disconf, 全称为 Distributed Configuration Management Platform (分布式配置管理平台), 专注于各种“分布式系统配置管理”的“通用组件”和“通用平台”, 提供统一的“配置管理服务”。

Disconf 如图 12-12 所示。

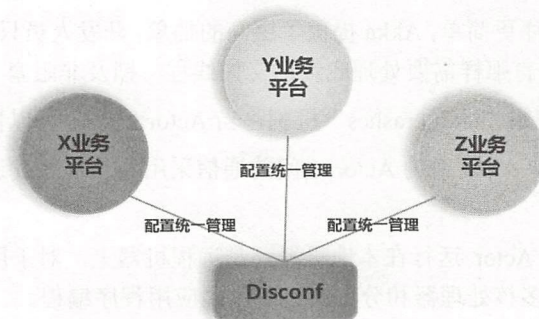


图 12-12 Disconf 分布式配置管理平台

项目的目标。

- 部署极其简单：同一个上线包，无须改动配置，即可在多个环境中（RD/QA/PRODUCTION）上线。
- 部署动态化：更改配置，无须重新打包或重启，即可实时生效。
- 统一管理：提供 Web 平台，统一管理多个环境（RD/QA/PRODUCTION）、多个产品的所有配置。
- 核心目标：一个 jar 包到处运行。

12.10.2 Apollo

Apollo（阿波罗）是携程框架部门研发的配置管理平台，能够集中化管理应用不同环境、不同集群的配置，配置修改后能够实时推送到应用端，并且具备规范的权限、流程治理等特性。

服务端基于 Spring Boot 和 Spring Cloud 开发，打包后可以直接运行，不需要额外安装 Tomcat 等应用容器。

Java 客户端不依赖任何框架，能够运行于所有 Java 运行时环境，同时对 Spring 环境也有较好的支持。

.NET 客户端不依赖任何框架，能够运行于所有 .NET 运行时环境。

配置界面如图 12-13 所示。

The screenshot displays the Apollo Configuration Center (Apollo 配置中心) interface. The main content area shows the configuration for an application named 'application'. The configuration is organized into a table with columns for '发布状态' (Release Status), 'Key ID', 'Value', '备注' (Remarks), '最后修改人 ID' (Last Modified By ID), '最后修改时间 ID' (Last Modified Time ID), and '操作' (Actions). The table lists various configuration items such as 'servlet', 'kibana.url', 'elastic.document.type', 'elastic.cluster.name', 'elastic.cluster', 'page.size', and 'zookeeper.address'. The interface also includes a sidebar with '环境列表' (Environment List) and '项目信息' (Project Information), and a top navigation bar with search and user options.

发布状态	Key ID	Value	备注	最后修改人 ID	最后修改时间 ID	操作
已发布	servlet	3000		song_s	2017-02-16 13:24:58	[编辑] [删除]
已发布	kibana.url	http://1.1.1.2:5600		song_s	2016-11-25 20:57:27	[编辑] [删除]
已发布	elastic.document.type	biz1		song_s	2017-01-11 19:14:06	[编辑] [删除]
已发布	elastic.cluster.name	es-cluster		song_s	2016-10-18 19:57:29	[编辑] [删除]
已发布	elastic.cluster	2.2.2-2-9300,4.4.4-9300		zhanglea	2016-12-09 14:19:43	[编辑] [删除]
已发布	page.size	20		song_s	2016-12-27 14:58:56	[编辑] [删除]
已发布	zookeeper.address	10.1.12.2		song_s	2016-10-19 11:33:50	[编辑] [删除]

图 12-13 Apollo 分布式配置中心

功能介绍

- 统一管理不同环境、不同集群的配置。
- Apollo 提供了一个统一界面集中式管理不同环境（environment）、不同集群（cluster）、不同命名空间（namespace）的配置。
- 同一份代码部署在不同的集群，可以有不同的配置，比如 ZK 的地址等。
- 通过命名空间（namespace）可以很方便地支持多个不同应用共享同一份配置，同时还允许应用对共享的配置进行覆盖。
- 配置修改实时生效（热发布）。

用户在 Apollo 修改完配置并发布后，客户端能实时（1 秒）接收到最新的配置，并通知到应用程序。

版本发布管理

所有的配置发布都有版本概念，从而可以方便地支持配置的回滚。

灰度发布

支持配置的灰度发布，比如点了发布后，只对部分应用实例生效，等观察一段时间没问题后再推给所有应用实例。

权限管理、发布审核、操作审计

应用和配置的管理都有完善的权限管理机制，对配置的管理还分为了编辑和发布两个环节，从而减少人为的错误。

所有的操作都有审计日志，可以方便地追踪问题。

客户端配置信息监控

可以方便地看到配置在被哪些实例使用。

提供 Java 和 .NET 原生客户端

提供了 Java 和 .NET 的原生客户端，方便应用集成。

支持 Spring Placeholder 和 Annotation，方便应用使用（需要 Spring 3.1.1+）。

同时提供了 HTTP 接口，非 Java 和 .NET 应用也可以方便地使用。

提供开放平台 API

Apollo 自身提供了比较完善的统一配置管理界面，支持多环境、多数据中心配置管理、权限、流程治理等特性。

不过 Apollo 出于通用性考虑,对配置的修改不会做过多限制,只要符合基本的格式就能够保存。

在调研中发现,对于有些使用方,它们的配置可能会有比较复杂的格式,如 XML、JSON,需要对格式进行校验。

还有一些使用方法如 DAL,不仅有特定的格式,而且对输入的值也需要进行校验后方可保存,如检查数据库、用户名和密码是否匹配。

对于这类应用, Apollo 支持应用方通过开放接口在 Apollo 中进行配置的修改和发布,并且具备完善的授权和权限控制。

部署简单

配置中心作为基础服务,可用性要求非常高,这就要求 Apollo 对外部依赖尽可能地少。

目前唯一的外部依赖是 MySQL,所以部署非常简单,只要安装好 Java 和 MySQL 就可以让 Apollo 运行起来。

Apollo 还提供了打包脚本,一键就可以生成所有需要的安装包,并且支持自定义运行时参数。

12.11 CAS

在分布式系统中,难免会有单点登录的需求。SSO 使得在多个应用系统中,用户只需要登录一次就可以访问所有相互信任的应用系统。

CAS 框架: CAS (Central Authentication Service) 是实现 SSO 单点登录的框架。

从结构上看, CAS 包含两个部分: CAS Server 和 CAS Client。CAS Server 需要独立部署,主要负责对用户的认证工作; CAS Client 负责处理对客户端受保护资源的访问请求,需要登录时,重定向到 CAS Server。

CAS Client 与受保护的客户端应用部署在一起,以 Filter 方式保护 Web 应用的受保护资源,过滤从客户端过来的每一个 Web 请求,同时, CAS Client 会分析 HTTP 请求中是否包请求 Service Ticket,如果没有,则说明该用户是没有经过认证的,于是 CAS Client 会重定向用户请求到 CAS Server。

然后是用户认证过程,如果用户提供了正确的 Credentials, CAS Server 会产生一个随机的 Service Ticket。缓存该 Ticket,并且重定向用户到 CAS Client (附带刚才产生的 Service Ticket), Service Ticket 是不可以伪造的。最后 CAS Client 和 CAS Server 之间完成了一个对用户的身份核实,用 Ticket 查到 Username。该协议完成了一个很简单的任务,所有与 CAS 的交互均采用 SSL 协议,确保 ST 和 TGC 的安全性。协议工作过程会有 2 次重定向过程,但是 CAS Client 与 CAS

Server 之间进行 Ticket 验证的过程对于用户是透明的。

总结如下：

- 访问服务——SSO 客户端发送请求访问应用系统提供的服务资源。
- 定向认证——SSO 客户端会重定向用户请求到 SSO 服务器。
- 用户认证——用户身份认证。
- 发放票据——SSO 服务器会产生一个随机的 Service Ticket。
- 验证票据——SSO 服务器验证票据 Service Ticket 的合法性，验证通过后，允许客户端访问服务。
- 传输用户信息——SSO 服务器验证票据通过后，传输用户认证结果信息给客户端。

CAS 认证的协议包括 CAS、OAuth、OpenID、SAML、REST。

应用场景包括：

- 分布式多系统用户集中管理；
- 用户权限集中管理；
- 多因素认证（如微信 PC 端登录手机确认）。

12.12 WebFlux

Spring 5 是流行的 Spring 框架的下一个重大的版本升级。Spring 5 中最重要改动是把反应式编程的思想应用到了框架的各个方面，Spring 5 的反应式编程以 Reactor 库为基础。

WebFlux 模块的名称是 spring-webflux，名称中的 Flux 来源于 Reactor 中的类 Flux。该模块中包含了对反应式 HTTP、服务器推送事件和 WebSocket 的客户端和服务端的支持。对于开发人员来说，比较重要的是服务器端的开发。在服务器端，WebFlux 支持两种不同的编程模型：第一种是 Spring MVC 中使用的基于 Java 注解的方式；第二种是基于 Java 8 的 Lambda 表达式的函数式编程模型。这两种编程模型只是在代码编写方式上存在不同。它们运行在同样的反应式底层架构之上，因此在运行时是相同的。WebFlux 需要底层提供运行时的支持，WebFlux 可以运行在支持 Servlet 3.1 非阻塞 I/O API 的 Servlet 容器上，或是其他异步运行时环境，如 Netty 和 Undertow。

最方便的创建 WebFlux 应用的方式是使用 Spring Boot 提供的应用模板。直接访问 Spring Initializr 网站 (<http://start.spring.io/>)，选择创建一个 Maven 或 Gradle 项目。

在 pom.xml 中加入相应的依赖：

```
<dependency>
```



```

<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-webflux</artifactId>
</dependency>

```

构建控制层，代码如下：

```

@RestController
public class BasicController {
    @GetMapping("/hello_world")
    public Mono<String> sayHelloWorld() {
        return Mono.just("Hello World");
    }
}

```

从代码中可以看到，使用 WebFlux 与 Spring MVC 的不同之处在于，WebFlux 所使用的类型是与反应式编程相关的 Flux 和 Mono 等，而不是简单的对象。对于简单的示例来说，这两者之间并没有什么太大的差别。对于复杂的应用来说，反应式编程和负压的优势会体现出来，可以带来整体的性能的提升。

那么对用户数据进行基本的 CRUD 操作又该如何处理呢？代码如下：

```

@Service
class UserService {
    private final Map<String, User> data = new ConcurrentHashMap<>();

    Flux<User> list() {
        return Flux.fromIterable(this.data.values());
    }

    Flux<User> getById(final Flux<String> ids) {
        return ids.flatMap(id -> Mono.justOrEmpty(this.data.get(id)));
    }

    Mono<User> getById(final String id) {
        return Mono.justOrEmpty(this.data.get(id))
            .switchIfEmpty(Mono.error(new
ResourceNotFoundException()));
    }
}

```

```

    Flux<User> createOrUpdate(final Flux<User> users) {
        return users.doOnNext(user -> this.data.put(user.getId(), user));
    }

    Mono<User> createOrUpdate(final User user) {
        this.data.put(user.getId(), user);
        return Mono.just(user);
    }

    Mono<User> delete(final String id) {
        return Mono.justOrEmpty(this.data.remove(id));
    }
}

```

类 `UserService` 使用一个 `Map` 来保存所有用户的信息，并不是一个持久化的实现。类 `UserService` 中的方法都以 `Flux` 或 `Mono` 对象作为返回值，这也是 `WebFlux` 应用的特征。在 `getById()` 方法中，如果找不到 ID 对应的 `User` 对象，会返回一个包含了 `ResourceNotFoundException` 异常通知的 `Mono` 对象。`getById()` 方法和 `createOrUpdate()` 都可以接受 `String` 或 `Flux` 类型的参数。`Flux` 类型的参数表示的是有多个对象需要处理。这里使用 `doOnNext()` 来对其中的每个对象进行处理。

类 `UserController` 是具体的 `Spring MVC` 控制器类。它使用类 `UserService` 来完成具体的功能。类 `UserController` 中使用注解 `@ExceptionHandler` 来添加了 `ResourceNotFoundException` 异常的处理方法，并返回 404 错误。`UserController` 的代码如下：

```

@RestController
@RequestMapping("/user")
public class UserController {
    private final UserService userService;

    @Autowired
    public UserController(final UserService userService) {
        this.userService = userService;
    }

    @ResponseStatus(value = HttpStatus.NOT_FOUND, reason = "Resource not found")
    @ExceptionHandler(ResourceNotFoundException.class)
    public void notFound() {

```



```

    }

    @GetMapping("")
    public Flux<User> list() {
        return this.userService.list();
    }

    @GetMapping("/{id}")
    public Mono<User> getById(@PathVariable("id") final String id) {
        return this.userService.getById(id);
    }

    @PostMapping("")
    public Flux<User> create(@RequestBody final Flux<User> users) {
        return this.userService.createOrUpdate(users);
    }

    @PutMapping("/{id}")
    public Mono<User> update(@PathVariable("id") final String id,
        @RequestBody final User user) {
        Objects.requireNonNull(user);
        user.setId(id);
        return this.userService.createOrUpdate(user);
    }

    @DeleteMapping("/{id}")
    public Mono<User> delete(@PathVariable("id") final String id) {
        return this.userService.delete(id);
    }
}

```

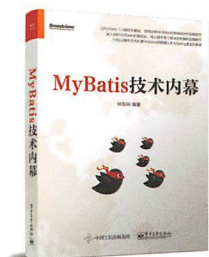
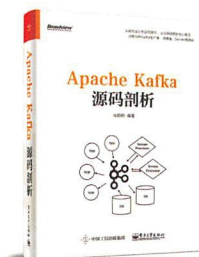
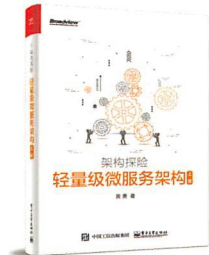
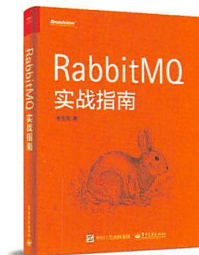
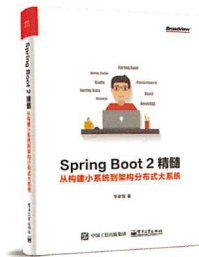
WebFlux 还支持基于 Lambda 表达式的函数式编程模型。与基于 Java 注解的编程模型相比，函数式编程模型的抽象层次更低，代码编写更灵活，可以满足一些对动态性要求更高的场景。不过在编写时的代码复杂度也较高，学习曲线也较陡峭。开发人员可以根据实际的需要来选择合适的编程模型。目前 Spring Boot 不支持在一个应用中同时使用两种不同的编程模式。

12.13 小结

作为全书的最后部分，本章列出了在微服务架构构建的过程中常用的一些开源组件，相互协调和配合使用这些组件，能够完成整体微服务框架的搭建。

其实其中提到的很多点，每一个点都可以使用一个章节甚至多个章节完整地进行描述和研究，但是本书的定位是让读者快速地了解微服务程序的构建，并不是对某个知识点进行深入的详解，所以，在工作中如果需要深入了解某个技术点，则可以根据书中的相关信息，继续深入研究和拓展。

好书分享



拒绝堆砌臃肿,支持纯正原创

欢迎投稿: chenxm@phei.com.cn

专家评价

本书的作者是业内资深架构师，有着多年的架构从业经验，常年活跃在Spring Cloud社区，致力于企业级微服务的解决方案和落地实现。全书以简洁、循序渐进的风格，由浅入深，从易到难，清晰地向读者介绍了微服务的技术思想和整个生态，值得一看。

美团无人配送部高级技术专家 刘审川

本书以实践为主，内容涵盖了微服务的整个生态，详细解释了实践微服务必须要面对的架构模式。本书系统性介绍了微服务的设计、开发、运维等各方面，结合了Spring Boot和Docker等热点技术，对微服务的整个生命周期做了全面介绍。本书适合对微服务实践感兴趣，以及想成为微服务架构师的人员阅读。

当当网技术总监 穆幽方

作者从基本概念谈起，循序渐进地介绍了各种实用技术，无论你处在技术链中的哪个环节，都能找到对应的知识点并快速学习，同时能了解其他环节的工作内容。本书值得技术爱好者收藏和学习。

中文在线运维经理 姜楠

随着微服务的兴起，我们可以看到微服务正逐步应用在企业中，其所带来的改变是不言而喻的，微服务的独立部署、可扩展性架构等特点可帮助企业提高生产效率、节省成本等。本书围绕一个中心，从微服务的概念、架构、管理、应用等方面层层深入，并通过翔实的案例结合实践进行展开，能够助力企业快速实施与部署微服务，值得一读。

柏睿数据科技（北京）有限公司产品总监 韩辉辉

作者张锋从实践中来到实践中去，将强大的微服务开发武器用简单易懂的方式展现在读者面前，值得广大技术人员深入领会。

丽华数据分析引擎工作室技术总监 吉更

本书作者张锋长期活跃在Spring Cloud社区，致力于帮助企业完成微服务改造，并提供咨询服务。本书作为全面的企业微服务架构实践指南，不仅可以帮助初识微服务的开发者完成微服务架构的搭建，也可以帮助正在实践微服务架构的团队解决实际问题。

北京环宇万维科技有限公司技术经理 苏忠亮

本书从微服务的思想基础、设计原则，延伸到Spring Boot、Docker、Spring Cloud及其他框架的介绍，不仅涉及微服务的自动化测试与质量管理、自动化部署、日志收集与监控，还提供完整的实战示例。全书清晰、透彻地剖析了微服务的整个生态，有助于读者快速提升对微服务的认知，构建自己的架构体系。

北京亿心宜行汽车技术开发服务有限公司技术部高软开发工程师 邓威



博文视点Broadview



新浪微博
weibo.com

@博文视点Broadview



责任编辑：陈晓猛
封面设计：李玲

上架建议：计算机>架构设计

ISBN 978-7-121-34342-1



定价：89.00元